Tarari

Content Processor Development Kit Agent Driver

Developer Guide

Legal Information

Tarari is a trademark or registered trademark of Tarari, Inc. or its subsidiaries in the United States and other countries.

Information in this document is provided in connection with Tarari products. No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted by this document. Except as provided in Tarari's Terms and Conditions of Sale for such products, Tarari assumes no liability whatsoever, and Tarari disclaims any express or implied warranty, relating to sale and/or use of Tarari products including liability or warranties relating to fitness for a particular purpose, merchantability, or infringement of any patent, copyright, or other intellectual property right. Tarari products are not intended for use in medical, life-saving, or life-sustaining applications. Tarari may make changes to specifications and product descriptions at any time, without notice.

Designers must not rely on the absence or characteristics of any features or instructions marked as "reserved" or "undefined." Tarari reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them.

* Other names and brands may be claimed as the property of others.

** Performance tests and ratings are measured using specific computer systems and/or components, and reflect the approximate performance of Tarari products as measured by those tests. Any difference in system hardware or software design or configuration can affect actual performance. Buyers should consult other sources of information to evaluate the performance of components they are considering purchasing. For more information on performance tests, and on the performance of Tarari products, contact us as indicated below.

Tarari Contact Information

Additional information: info@tarari.com

Internet: http://www.tarari.com/

Telephone: (858) 385-5131

Fax: (858) 385-5129

Office hours: 8 A.M. to 5 P.M. Pacific Time

Documentation questions or comments: documentation@tarari.com

Tarari, Inc.
10908 Technology Place
San Diego, CA 92127-1874
USA

A02209-001

Contents

Concepts

Introduction

This document describes:

- The memory management architecture for the Double Data Rate Synchronous Dynamic Random Access Memory (DDR SDRAM) resources on the Tarari Content Processing Platform (CPP) board
- A Standard Agent Interface (SAI), that drives this memory management architecture
- The interface between the CPP Base Device Driver (BDD) and Agent Device Drivers (ADDs)
- The configuration interface for the BDD
- A sample interface between an ADD and a user application

Both the Anti-Virus and XML acceleration Agents incorporate this memory scheme, and the SAI, for communications between hardware and software. Future development efforts, including those of third parties using the Development Kit, must follow this architecture to maintain compatibility and reduce the time to market.

Hardware and software engineers who develop Agents for the CPP board must thoroughly understand the contents of this document, and its implications, before undertaking any new development effort.

This document assumes that you, as the developer, know how to write Device Drivers in the environment of your choice.

For CPP board installation instructions, refer to the *Content Processing Platform Installation Guide*, Tarari part number A02202-001, or later.

Terms

Table lists terms and acronyms used in this document.

Term	Description
ABI	Agent Bus Interface
ADD	Agent Device Driver: The CPP has a fundamental device driver that can be accessed directly by user-space applications. However, an ADD is layered on top of the basic device driver to provide an efficient means of customizing an Agent's interface.
Agent	Acceleration Agent (Anti-Virus, XML, and the like). Each CPE can run up to two Agents.
API	Application Program Interface: A set of interface points to a software resource
BDD	Base Device Driver
Bitstream	A continuous flow of binary digits (bits), through some form of communication (in this case, the PCI bus), with no break or separations between the characters. A bitstream contains the Agent Set, which is the information needed to install one or two Agents on the CPEs.
CCL	Configuration and Control Logic: Logic internal to the CPC that manages reconfiguration of the Agent CPEs
CPC	Content Processing Controller: A logic component on the CPP board that acts as a bridge or arbiter between the PCI bus, Agents, and the DDR SDRAM
CPE	Content Processing Engine: One of two reconfigurable logic components on the CPP board. Each CPE can run up to two Agents.
CPP	Tarari's Content Processing Platform, and the board on which it is installed
DDR SDRAM	Double Data Rate Synchronous Dynamic Random Access Memory: The CPP board's main memory.
DLL	Dynamic Link Library: Executable code module that can be loaded on demand and linked at run time, then unloaded when the code is no longer required.
DMAC	Direct Memory Access Controller: The DMAC is a logical component of the CPC. It transfers data between the CPP and system memory.
Dword	A dword (double word) pointer points to 4 bytes of data at a time. For example: Pointer X initially points to bytes 0-3. Once it increments by 1, Pointer X points to bytes 4-7.
EOF	End of File
FIFO	First in, first out
IOCTL	Input/Output Control
ISR	Interrupt Service Routine
OS	Operating System
KLM	Kernel Loadable Module: An extensible interface under Linux that dynamically inserts code into the operating system kernel at runtime. Most device drivers are implemented as KLMs.
LOF	Length of File
OS	Operating System (typically Windows* or Linux)
Qword	A qword (quad word) pointer points to 8 bytes of data at a time. For example: Pointer Y initially points to bytes 0-7. Once it increments by 1, Pointer Y points to bytes 8-15.
SAI	Standard Agent Interface
SOF	Start of File
SRAM	Static Random Access Memory: Low latency memory on the CPP board. Each Agent CPE has two associated SRAM banks.
UA	User Application
TMU	Task Management Unit

Table 1: Terms Used in this Document

CPDK Overview

The Tarari Content Processor Development Kit (CPDK) is a combination of hardware and software content processing building blocks that creates a flexible platform designed to accelerate a variety of compute-intensive algorithms. The CPDK leverages the standard PCI bus and operating system (OS) interfaces to facilitate easy integration into servers and network appliances. The CPDK also includes development tools and sample code designed to help developers to quickly learn the technology.

Figure 1 shows how software applications can transport data that requires compute-intensive operations into the hardware domain for accelerated processing.

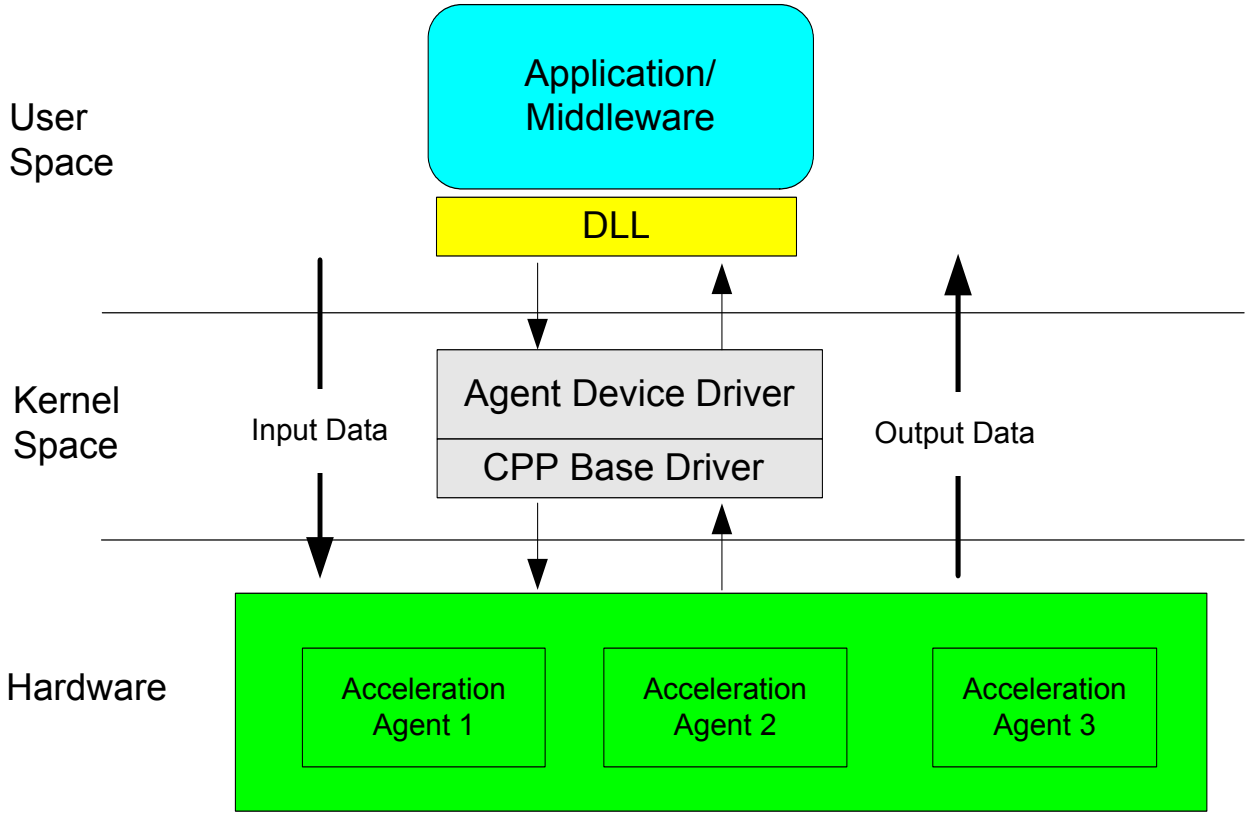

Figure 1: Processing Stack

In a typical CPDK project:

- **Software developers** write an Agent Device Driver and a DLL (or equivalent), to connect the application to the CPP Base Driver.
- **Hardware developers** design the targeted hardware Agent(s), using a CPE design methodology.

At a lower level, software communicates with hardware by passing data from host memory to the DDR SDRAM on the CPP board. Figure 2 shows the CPP board's resources and interconnections. The CPC contains a Direct Memory Access Controller (DMAC) to move data from the PCI Bus to the DDR SDRAM at high data rates. Data flows from the DDR SDRAM to one of the Agents, loaded into a Content Processing Engine (CPE), using the CPC and the Agent Bus. After processing, the CPC moves data back to the DDR SDRAM. The DMAC moves data back to host memory using the PCI Bus.

For software to hardware out-of-band communications, the CPC uses a 32-bit write-only register for each Agent. For hardware to software out-of-band communications, the CPC uses a second 32-bit read-only register for each Agent. To interrupt the software, Agents can load a third 32-bit interrupt status register.

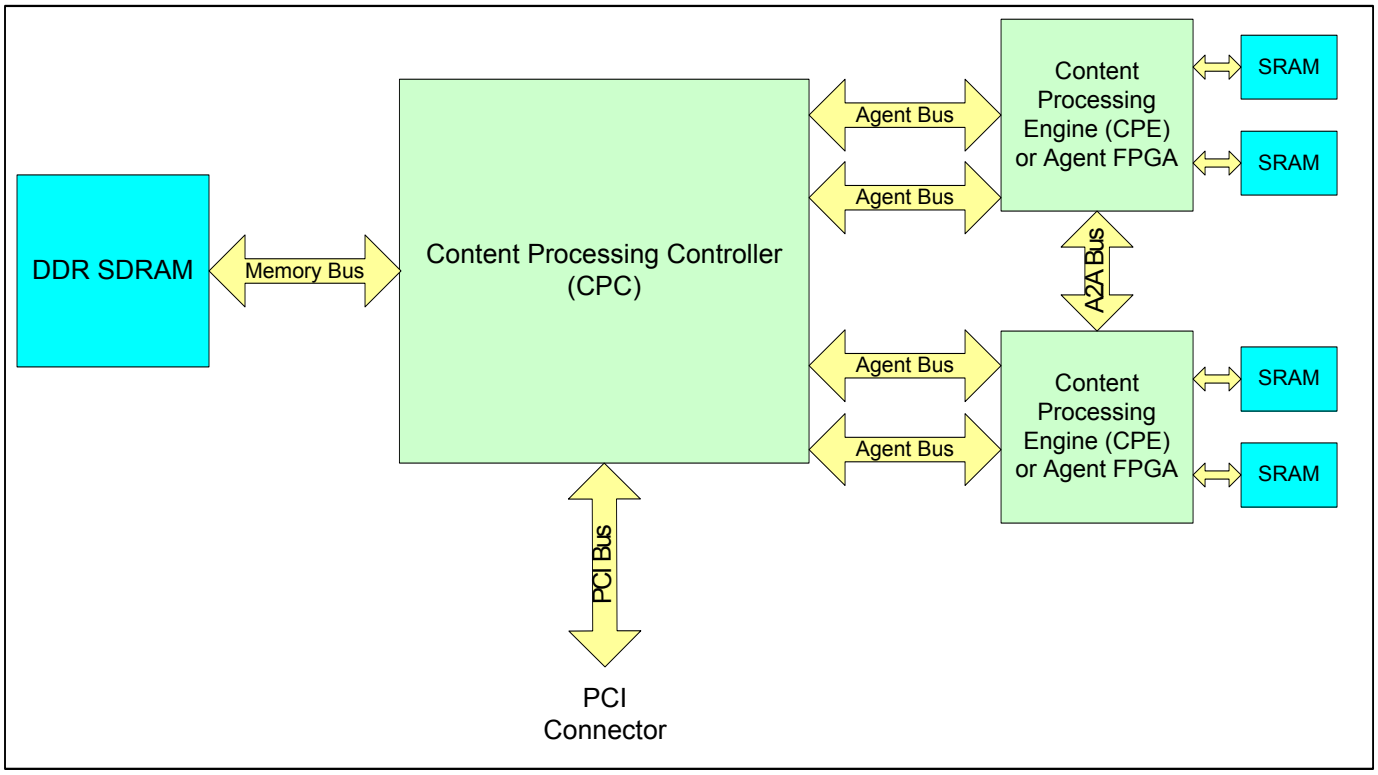

Figure 2: CPP Board Block Diagram

The basic CPP hardware/software interface defines a single out-of-band command. This command loads the Agent's DDR SDRAM Base Address. During Agent initialization, the CPP Base Driver uses this command to communicate its memory base address. The Agent then accesses DDR SDRAM, using a physical address it calculates by adding the Base Address to its local address. Hardware and software communicate using the local address.

During Agent initialization, the CPP Base Driver copies the Agent's total allocation of DDR SDRAM to the Device Driver.

After Agent initialization, the Device Driver and Agent use the DDR SDRAM for all in-band high bandwidth data communications. The Base Driver and CPP Architecture place no further constraints upon this hardware and software pair, other than the base address of memory (known only to hardware), and the total allocation of memory (known only to software).

Agent Interface Compatibility

Because the base architecture places few constraints on the DDR SDRAM, Agent designers are free to use the memory in a way best suited for their specific application. While this might lead to optimum memory efficiency and perhaps even optimal memory bandwidth use, it can present two problems:

- **Each Agent requires the design and development of a memory use scheme**. This leads to unnecessary re-design of hardware or software interfaces for (at best) marginal gains in performance and efficiency. Most of CPDK's added value is the Agent itself, not the hardware or software interface.
- **Agents cannot communicate directly with each other**. This severely limits performance, because it prohibits process chaining due to incompatibilities between one Agent and another. There are many applications in which new Agents can be installed in the CPEs to further process the results of an existing Agent. Without a consistent memory use architecture, a round trip between host memory and DDR SDRAM is required for each Agent. This increases latency, and unnecessarily consumes PCI bandwidth.

Introduction to the Agent Bus Interface

The Agent Bus Interface (ABI) is a low-level interface between the CPC and each CPE, as Figure 3 on the next page shows. Its operation is quite complicated, and subject to periodic updates to improve product performance.

For ease of programming, and to ensure future compatibility, we recommend that you focus your programming efforts on the Standard Agent Interface (SAI) instead, as described on the next page.

Introduction to the Standard Agent Interface

This section introduces the Standard Agent Interface (SAI). The SAI provides a consistent hardware and software interface for both in-band and out-of-band communications. Figure 3 shows how the SAI inserts into the existing hierarchy.

- From the **software** perspective, Device Drivers can re-use code that communicates with the Base Driver for initializing, building, and moving data to and from the standard data structures.
- From a **hardware** perspective, Agent developers receive and send data using a simple first in, first out (FIFO) interface.

The SAI module hides the memory management functions required to communicate with software, and to transfer data between the Agent and DDR SDRAM. Agent developers can focus on the valuable features of their processing accelerator, rather than low value, but necessary, interface management.

For more details on the SAI, see “Standard Agent Interface” on page 38.

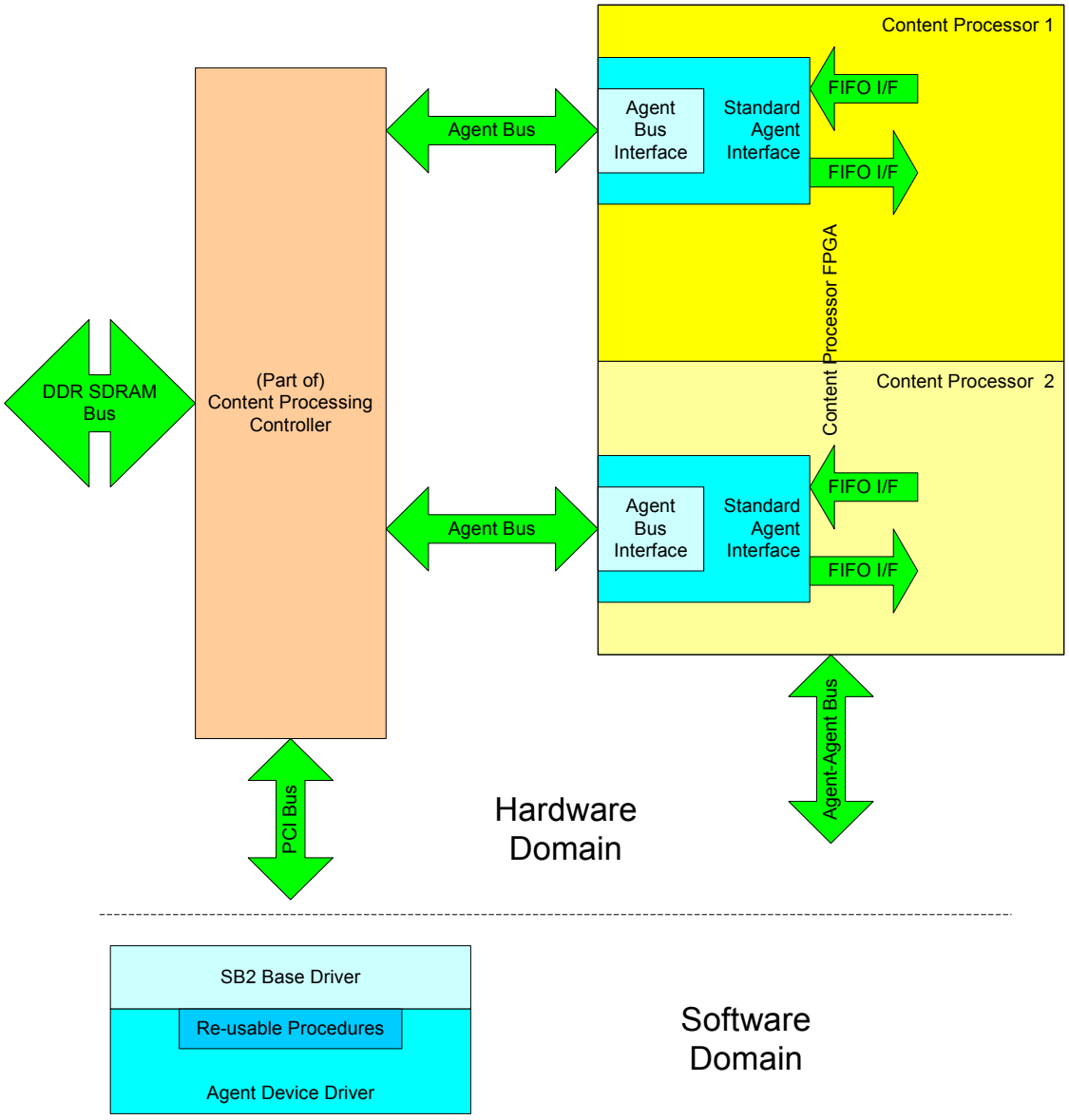

Figure 3: Hardware/Software Implications of the SAI

Content Processing Platform

The CPP board is a hardware accelerator designed to offload and accelerate key compute-intensive algorithms or processes performed by servers or appliances in a network. Developers can use it to target applications for hardware acceleration, with significant performance improvement.

Figure 4 shows the main CPP board components.

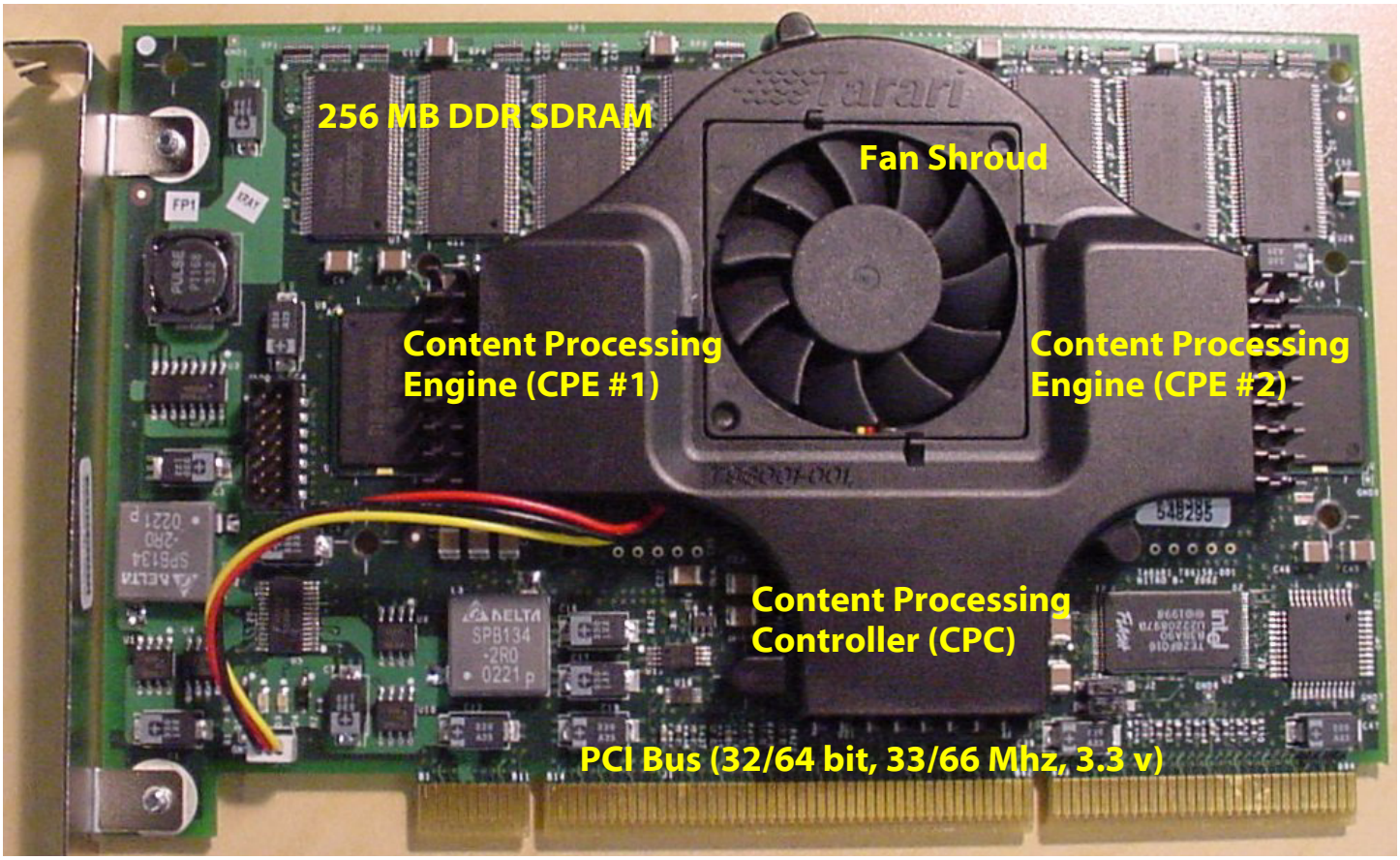

Figure 4: CPP Board Components

The CPP board contains two dynamically configured Content Processing Engines (CPEs). You can configure these CPEs at system startup using a script, or at runtime, without rebooting. After configuration, each CPE is separated into sub-components called Agents.

Each CPE can run one or two Agents, depending upon the memory requirements. You can program each Agent to perform certain functions, such as RSA, DES, PRNG, and many other algorithms.

The CPP board also features 256 MB of DDR SDRAM. This memory shares data between the CPEs and your applications.

Detailed component descriptions follow on the next page.

Content Processing Controller

The CPC:

- Provides transparent access to:
  - High bandwidth, high capacity shared memory (DDR SDRAM)
  - High-speed standard bus
- Configures the CPP with acceleration Agent sets:
  - Supports multiple concurrent independent acceleration Agents
  - Supports entire acceleration Agent sets with dependent Agents
- Simplifies **managing** software algorithms in hardware

Content Processing Engines

Each CPE:

- Accelerates algorithms through runtime hardware configurations
- Provides simple and powerful interface to all CPP resources:
  - Low latency local memory (SRAM)
  - The CPC, shared memory, and system bus
- Simplifies **implementing** software algorithms in hardware
- Supports one or two Agents, as required

Software Model

CPP Data Flow

The CPP data flow model consists of multiple hardware and software layers. At the lowest layer, a Base Device Driver (BDD) communicates directly with one or more CPP boards in the system. Applications can interface with multiple Agent Device Drivers (ADDs), but each Agent you configure on the CPP requires at least one corresponding ADD.

Figure 5 shows the CPP data flow from the user application, through the ADD, the BDD, the hardware layer, and back to the application again.

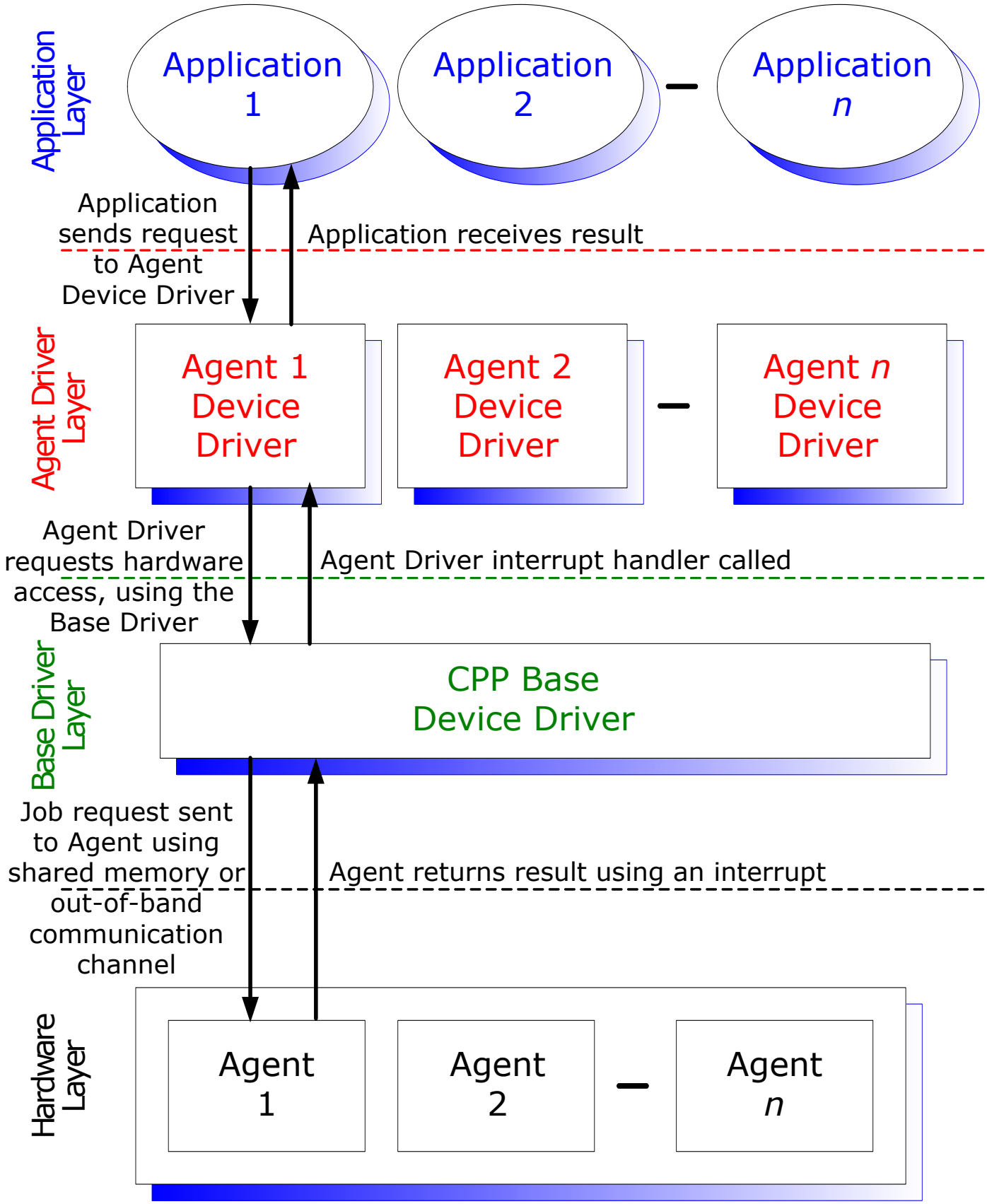

Figure 5: CPP Data Flow Diagram

CPP Reconfiguration Model

The BDD must **always** be loaded on the system, but ADDs can load or unload at runtime as needed. Loading an ADD does not affect other installed ADDs, although having more ADDs implies that there is more competition for CPP resources, which could impact performance.

An ADD might load even if its corresponding Agent is not configured at the time of the load. The BDD alerts ADDs of the insertion or removal of Agents as new Agent sets are configured onto the CPP. The ADDs are then responsible for registering or un-registering the affected Agents, as required.

The BDD provides an API to manage acceleration Agent set configuration. This API consists of writing Agent sets to a cache on the CPP, and configuring a CPE from a cached Agent set. Each CPP reserves a portion of its DDR SDRAM for storing acceleration Agent sets. The current architecture supports up to eight concurrent independent Agent sets. Once an Agent set is stored on the board, the API might request that the CPP configure a CPE using the specified Agent set.

Tarari provides an Acceleration Agent Set Management software application that exercises the acceleration Agent set management interface. In addition, Tarari will publish the management API in the future, for third party applications to directly manage Agents. To minimize conflicts, only one application can access the configuration API at a time.

DDR SDRAM is allocated to ADDs in contiguous page sized segments. If a memory segment is not available, the ADD is notified, and it has to wait until enough memory frees up before it proceeds. After the ADD successfully allocates memory, the ADD cannot access it directly. Instead, it must call functions declared in the BDD that read and write to its allocated memory. This ensures the ADDs do not read or write outside the memory segment allocated to them.

NOTE: If the Raw Access Calls are implemented in the future, the ADD will have direct memory access. For more information, see “Raw Access Calls” on page 26.

After the BDD loads, it allocates a contiguous block of memory that it uses to store CPE bitstreams. The loading of these bitstreams and programming of the CPEs is initiated by the usual system calls. Each bitstream contains a header that the BDD can read. This header information is stripped from the bitstream and saved before writing the bitstream to memory. Your application can query the header information to identify which bitstreams are loaded and programmed in the CPEs.

The BDD arbitrates between the ADDs and their access to the CPP board, by allowing the ADD with the highest priority to access the device and blocking all others. If two or more ADDs have the same priority level, then a round robin scheduling scheme is used on those ADDs.

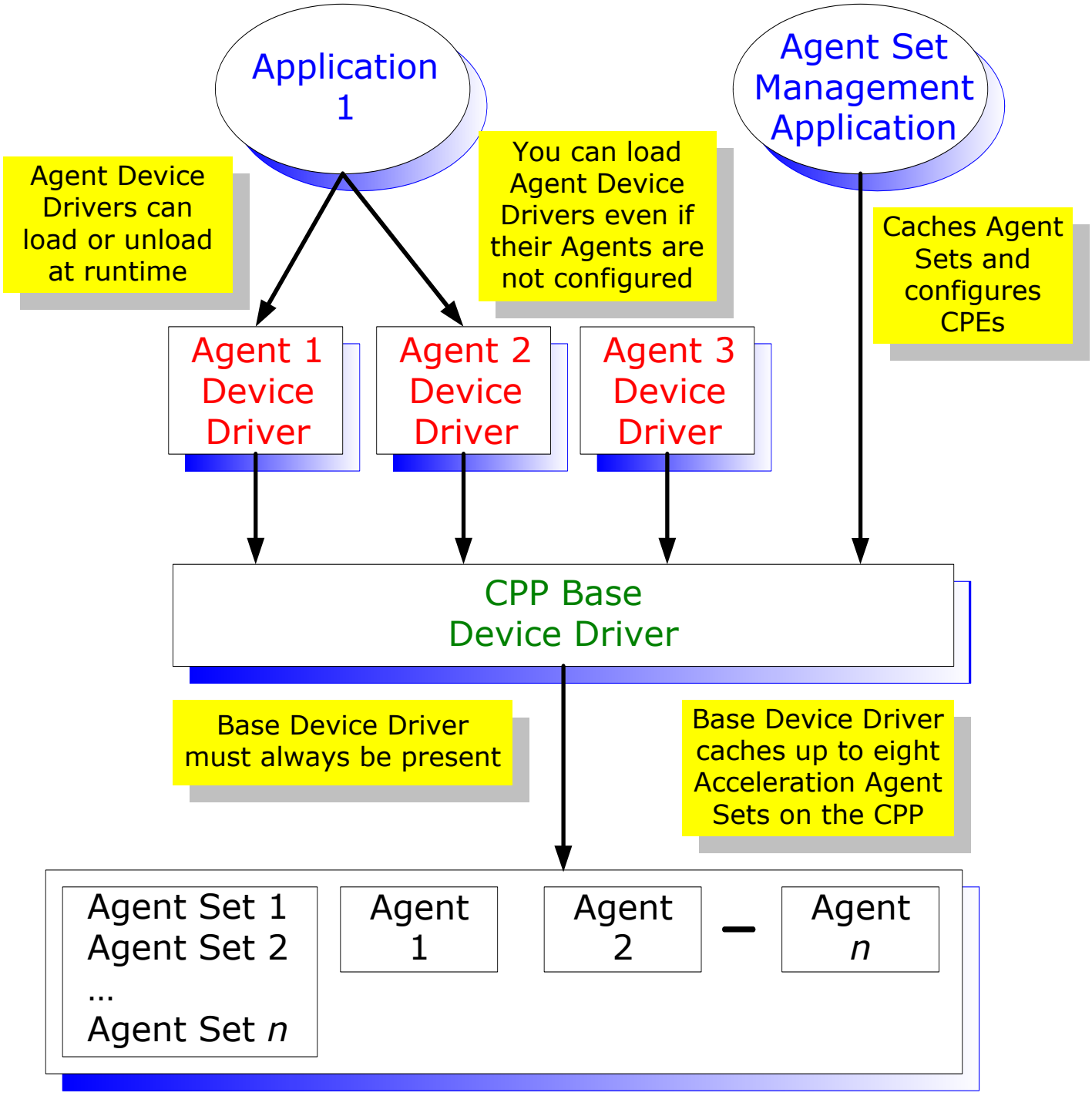

Figure 6: CPP Reconfiguration Model

CPP Base Device Driver (BDD)

The BDD manages DDR SDRAM on the CPP boards, the configuration of the CPEs, and the synchronization between the ADDs.

CPP Agent Device Drivers (ADDs)

ADDs are custom-made for every Agent, but every ADD must follow a protocol to communicate with the BDD. Each ADD must initially register itself with the BDD by informing it of the type or types of Agents it accesses, and the size of the memory segment it wants to allocate. The ADD must also register an interrupt service routine (ISR) with the BDD.

- If the **registrations succeed**, the ADD is assigned a handle. The ADD uses this handle to access the memory segment allocated to it by the BDD. ADDs are free to manage the memory allocated to them in any way.
- If the ADD is **no longer needed** and it is about to unload, it must un-register itself with the BDD. This directs the BDD to free the memory segment formerly allocated to the ADD.

CPP Example

Before an application can use the Agents, an ADD for each type of Agent the application accesses must be loaded in the system. This ADD handles all Agents of the same type in the CPEs, and across multiple CPP boards. For example, consider the two-board application in Figure 7:

- Agents 1 and 2 of CPE 1 on CPP Board 1 are configured for RSA.
- Agents 1 and 2 of CPE 2 on CPP Board 1 are configured for DES.
- Agent 1 of CPE 1 on CPP Board 2 is configured for RSA.
- Agent 2 of CPE 1 on CPP Board 2 is configured for XML.
- CPE 2 on CPP Board 2 is not configured (NC).
- Three ADDs must be loaded in the system: one for RSA, one for DES, and one for XML.
- The applications access the Agents using the ADD, layered on top of the BDD.

The Agents in Figure 7 are examples only. Your application might use different Agent and CPE assignments.

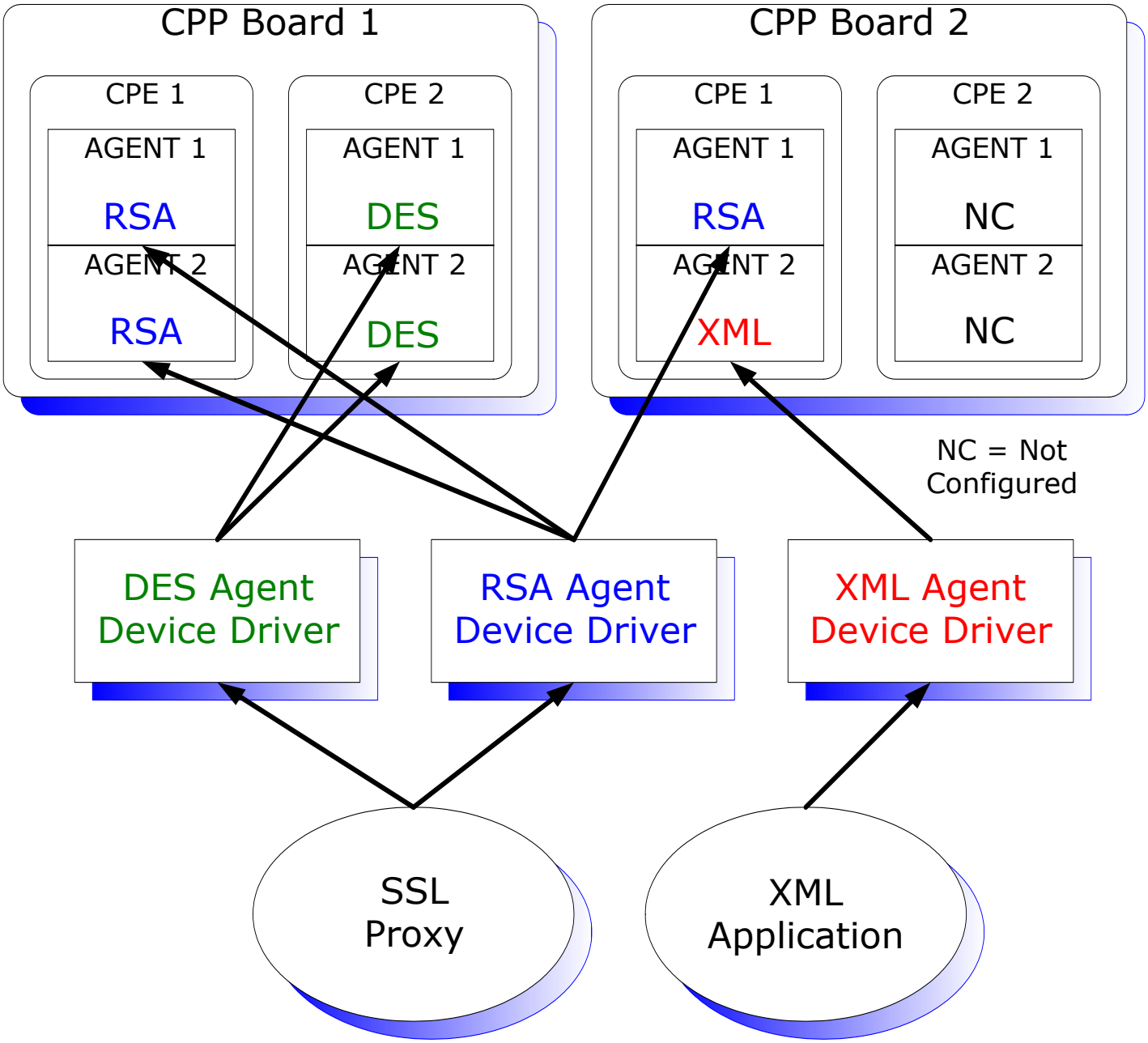

Figure 7: Agents on Two CPP Boards

BDD Responsibilities

- **CPE Configuration**: The BDD keeps track of every bitstream loaded into the system and programmed into each CPE on each CPP board, and provides an interface for applications to manage configurations.
- **Interrupts**: The BDD handles all interrupts from each CPP board, and routes them to the correct ISR in a particular ADD.
- **DDR Memory Management**: The BDD allocates segments of memory to ADDs, and performs bounds checking on any read or write access to the allocated memory.

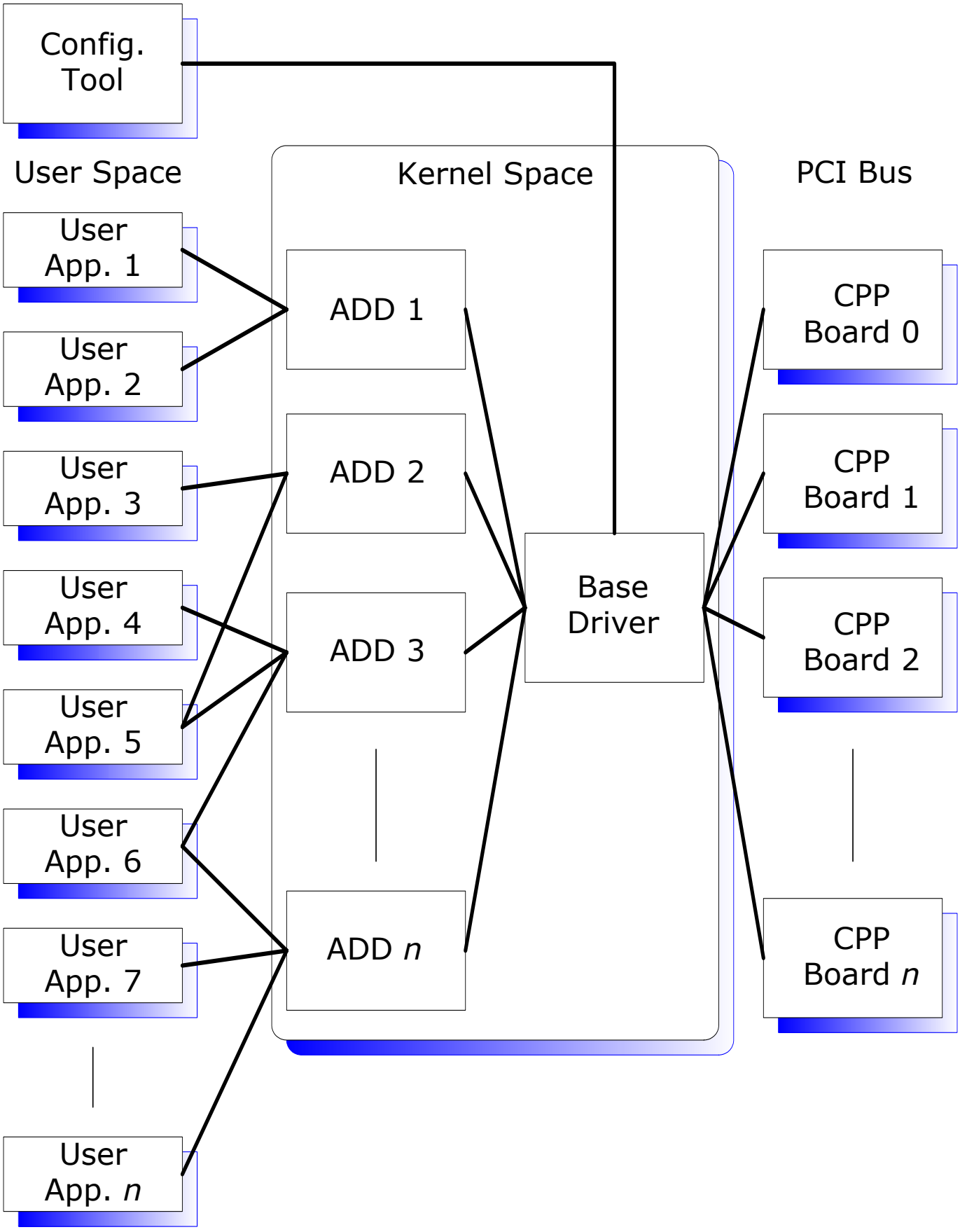

Figure 8: User Applications and Device Drivers

BDD to ADD Interface

Overview

The Base Device Driver (BDD) provides these API calls to communicate with the Agent Device Drivers (ADDs):

- **Registration calls and callbacks** direct an ADD to request access to particular types of Agents and instantiations of those Agents.
- **Data Transfer calls** provide read and write access to an Agent's I/O port and shared device memory. The BDD provides direct memory access (DMA) capabilities. Your application can improve DMA performance if the BDD accesses cached address mappings to application buffers.
- **Performance Optimization calls** currently provide the necessary steps for implementing DMA transfers, but might have additional options in the future.
- **Raw Access calls** provide the raw virtual address of an Agent's address space to an ADD. Your application can use this address to implement Linux memory map (mmap) functionality within the ADD. Although we recommend that ADDs access Agent memory using direct memory access for performance reasons, mmap is quite useful when developing and debugging new Agents.

Data Structures

The CPP data structures and APIs are designed to be primarily architecture independent. However, some structures are specific to Windows* or Linux environments.

Architecture Dependent

For more information on the architecture-dependent structures in Table 2, see the Linux and Windows CPP sections in this document.

Architecture Independent

Table 2 lists the architecture independent structures.

Structure	Item	Description
CppAgtID_t	Purpose	Each Agent is assigned a unique Agent identifier value during Agent creation. The BDD uses this to match an Agent with its associated ADD. This structure contains an Agent identifier (32-bit) value.
	Syntax	typedef **CppAgtID_t** uint32_t;
CppReturnCode_t	Purpose	The BDD to ADD interface uses these codes, defined by this structure, for return values.
	Syntax	typedef enum **CppReturnCode** { CPP_SUCCESS = 0, CPP_NO_AGENT, CPP_NO_RESOURCES, CPP_INVALID_PARAMETER, CPP_INVALID_AGENT_HANDLE, CPP_AGENT_ID_NOT_AVAILABLE } **CppReturnCode_t**;
CppAddrType_t	Purpose	User space and kernel space use different address spaces, but the BDD must transfer data to and from buffers that might reside either in user space applications or kernel space drivers. As a result, data transfer calls must indicate the address space to which a particular buffer refers.
	Syntax	typedef enum **CppAddrType** { USER_SPACE_VIRTUAL_ADDR, KERNEL_SPACE_VIRTUAL_ADDR } **CppAddrType_t**;
CppAgtDrvCbackMsg_t	Purpose	• The BDD uses a callback routine to notify an ADD whenever an Agent related to that ADD is added or removed from the system. This structure type indicates which event occurred and instigated the callback. • The CPP_ADD_AGENT message indicates that a new Agent is now available for registration. The BDD alerts the ADD whose Agent identifier matches that of the newly configured Agent. • The CPP_RELINQUISH_AGENT message warns an ADD that the BDD received a request to overwrite that Agent with a different Agent set. The BDD allows the ADD time to clean up all its outstanding requests, and waits for an acknowledgement indicating that the Agent is clear for removal. If the ADD does not respond with an acknowledgement within a timeout period, the BDD sends the CPP_REMOVE_AGENT message to the offending ADD to forcibly reconfigure the CPE. If an ADD receives the CPP_REMOVE_AGENT message, it might not access the Agent's resources.
	Syntax	typedef enum **CppAgtDrvCbackMsg** { CPP_ADD_AGENT, CPP_RELINQUISH_AGENT, CPP_REMOVE_AGENT } **CppAgtDrvCbackMsg_t**;

Table 2: Architecture Independent Data Structures

Registration Calls and Callbacks

Table 3 lists the registration calls and callbacks.

Function Call	Item	Description

CppRegisterAgentID	Purpose	• Each Agent is assigned a unique identifier during the "Agent Signing" process after the Agent is created. This function call lets ADD request access to all Agents with the specified identifier. Each Agent identifier might only have one associated ADD. As a result, this call acts as a gate for all other Agent interface calls. Successful calls to this function do not necessarily mean that the ADD has registered for particular instantiations of an Agent. This is accomplished by calling CppRegisterAgent().
		• On a successful call, the CPP_SUCCESS value is returned. If another ADD already has exclusive access to the requested Agent identifier, CPP_AGENT_ID_NOT_AVAILABLE is returned.
		• On success, the NumConfiguredAgts field is updated with the number of Agents of the specified identifier that already are configured. For example, if the Diagnostic ADD requests access to the Diagnostic Agent Identifier and two Diagnostics Agents are already configured on a CPP device, the NumConfiguredAgts field is set to the value 2. If no Agents of the specified type are configured (NumConfiguredAgts == 0), the ADD cannot access an Agent until it is notified by the AgtDrvCallback() function that an appropriate Agent has been configured.
		• The BDD calls the AgtDrvCallback() routine when any of these CppAgtDrvCbackMsg_t messages are sent to the ADD:
		  • CPP_ADD_AGENT: A new Agent of the specified type is available
		  • CPP_RELINQUISH_AGENT: Another Agent has requested the CPE resources currently in use by one or more registered Agents. In this case, the ADD must shut down all connections to the Agent in question, and acknowledge that the resources are available to the BDD using the CppAckAgtRemoval() call.
		  • CPP_REMOVE_AGENT: If the ADD does not acknowledge a request to free Agent resources, the BDD times out, and calls the offending ADD, indicating that the Agent's resources have been revoked. This is not yet implemented in Linux.

	Syntax	CppReturnCode_t **CppRegisterAgentID** (CppAgtID_t agtID, uint32_t* numConfiguredAgts, void(*agtDrvCallback) (CppAgtDrvCbackMsg_t agtMsg, CppAgtID_t agtID)
	Inputs	• agtID: The unique Agent identifier assigned to the Agent during its creation and signing • numConfiguredAgts: This returns how many of the specified Agents currently are configured • agtDrvCallback: The BDD makes this callback when Agents are added or removed from the system

Table 3: Registration Calls and Callbacks

Function Call	Item	Description
CppRegisterAgentID (continued)	Returns	• CPP_SUCCESS: The registration for the AgentID identifier succeeded. The ADD now has the ability to register for particular instantiations of that AgentID and communicate with those Agents. • CPP_AGENT_ID_NOT_AVAILABLE: Another ADD has already requested exclusive access to all Agents with the specified identifier. • CPP_INVALID_PARAMETER: agtDrvCallback() is NULL
CppUnregisterAgentID	Purpose	Releases the resources acquired in CppRegisterAgentID()
	Syntax	CppReturnCode_t **CppUnregisterAgentID** (CppAgtID_t agtID)
	Inputs	• agtID: The unique Agent identifier assigned to the Agent during Agent creation and signing
	Returns	• CPP_SUCCESS: The ADD has successfully relinquished access to all Agents with an identifier matching agtID. • CPP_NO_ACCESS: The call fails with this value if the identifier is not registered.
CppRegisterAgent	Purpose	• This call requests access to one instance of an Agent whose identifier matches agtID and whose memory requirements are agtMemSize bytes. Upon successful completion, the call sets the CppAgtHandle_t, pointed to by agtHandlePtr, to a valid handle. It uses this handle in future calls to the BDD to reference the newly registered Agent. • The call fails unless the calling ADD has already received access to all Agents of this type using the CppRegisterAgentID() call.
	Syntax	CppReturnCode_t **CppRegisterAgent** (CppAgtID_t agtID, uint32_t agtMemSize, CppAgtHandle_t* agtHandlePtr, void (*interruptHandler) (CppAgtHandle_t agtHandle, uint32_t interruptData))
	Inputs	• agtID: The unique Agent identifier assigned to the Agent during Agent creation and signing • agtMemSize: The amount of device memory that the Agent requires to function • agtHandlePtr: The BDD assigns a handle to the ADD upon a successful registration • interruptHandler: The BDD calls this ISR when the associated Agent generates an interrupt. The ISR is told which Agent generated the interrupt using the agtHandle parameter. The interruptData parameter holds the 32-bit interrupt data.
	Returns	• CPP_SUCCESS: The ADD has successfully registered the Agent • CPP_NO_AGENT: No Agents are available for registration • CPP_INVALID_PARAMETER: One of the parameters is outside an acceptable range. For example: If the interrupt handler is NULL, or the Agent's memory size is negative, the routine returns with an invalid parameter message. • CPP_NO_RESOURCES: Too many ADDs are registered, and no available slots are left or a memory allocation for the Agent fails

Table 3: Registration Calls and Callbacks (continued)

Function Call	Item	Description
CppUnregisterAgent	Purpose	Releases the Agent resources associated with the Agent reference by agtHandle
	Syntax	CppReturnCode_t **CppUnregisterAgent** (CppAgtHandle_t agtHandle)
	Inputs	agtHandle: A valid handle to an Agent owned by the calling ADD
	Returns	• CPP_SUCCESS: The ADD has successfully unregistered the Agent • CPP_INVALID_AGENT_HANDLE: No Agents correspond to the handle given
CppAckAgtRemoval	Purpose	• After an ADD receives access to an Agent identifier using the CppRegisterAgentID() call, the BDD alerts the Agent if any Agents matching that identifier are added or removed from the system. This notification occurs through the callback routine given as a parameter during the CppRegisterAgentID()process. • If the ADD receives the CPP_RELINQUISH_AGENT message through its callback routine, it must shut down all connections to the specified CppAgtHandle_t and free any resources associated with the Agent. After the ADD has freed such resources, it sends an acknowledgement back to the BDD through the CppAckAgtRemoval() call indicating that the Agent is now available for removal. • After acknowledging the agent's removal, the ADD must not access any resources associated with the agent. Furthermore, if the ADD does not send an acknowledgement back to the Base Driver within a TBD timeout period, the Base Driver will send a CPP_REMOVE_AGENT command to the offending ADD and forcefully remove the agent.
	Syntax	CppReturnCode_t CppAckAgtRemoval (CppAgtHandle_t agtHandle)
	Inputs	agtHandle: A valid handle to an agent owned by the calling ADD
	Returns	• CPP_SUCCESS: ADD successfully acknowledged the agent removal • CPP_INVALID_AGENT_HANDLE: No agents correspond to the given handle

Table 3: Registration Calls and Callbacks (continued)

Data Transfer Calls

Table 4 lists the Data Transfer Calls.

Function Call	Item	Description
CppWriteRegister	Purpose	Each agent has a dedicated 32-bit I/O port register it uses for communication of out-of-band information with its associated ADD. Typically, commands are written to the register, and the status is read back from the register. The format for such commands and status are largely agent-specific. The only restriction is that the most significant bit of the command register is reserved for agent initialization by the BDD, and no agent-specific command can ever set bit 31 of the register.
	Syntax	CppReturnCode_t **CppWriteRegister** (CppAgtHandle_t agtHandle, uint32_t command)
	Inputs	• agtHandle: A valid handle to an agent owned by the calling ADD • command: 32-bit command sent to the agent referenced by agtHandle
	Returns	• CPP_SUCCESS: The ADD successfully wrote the command to the agent's 32-bit I/O port • CPP_INVALID_AGENT_HANDLE: No agents correspond to the handle given
CppReadRegister	Purpose	Each agent has a dedicated 32-bit I/O port register used for communication of out-of-band information with its associated ADD. Typically, commands are written to the register, and the status is read back from the register.
	Syntax	CppReturnCode_t **CppReadRegister** (CppAgtHandle_t agtHandle, uint32_t* status)
	Inputs	• agtHandle: A valid handle to an agent owned by the calling ADD • command: 32-bit command sent to the agent referenced by agtHandle
	Returns	• CPP_SUCCESS: The ADD successfully read the command from the Agent's 32-bit I/O port • CPP_INVALID_AGENT_HANDLE: No agents correspond to the handle given

Table 4: Data Transfer Calls

Function Call	Item	Description
CppDirectWriteMem	Purpose	• ADDs can write directly to an agent's device memory. For performance reasons, the preferred method of data transfer is to DMA data directly to the board. However, for small data transfers it might be more efficient to directly access board memory rather than paying the overhead of setting up a DMA transaction. • The CppDirectWriteMem() call writes length bytes to the specified agent's memory buffer, starting at an offset of agtMemOffset bytes from the agent's first memory location. The source of the data is sourceBuffer. • The addrType indicates what type of address sourceBuffer is. It can take any of the CppAddrType_t values, which address the possible virtual and physical memory segments.
	Syntax	CppReturnCode_t **CppDirectWriteMem** (CppAgtHandle_t agtHandle, CppAddrType_t addrType, uint8_t* sourceBuffer, uint32_t agtMemOffset, uint32_t length)
	Inputs	• agtHandle: A valid handle to an agent owned by the calling ADD • addrType: Type of address found in sourceBuffer. The source data might reside either in a user-space virtual buffer or a kernel buffer. • sourceBuffer: Pointer to the start of the source data to be sent to the device • agtMemOffset: Device destination location given as a byte offset into an agent's shared memory segment • length: Number of bytes to transfer to the agent's shared memory segment
	Returns	• CPP_SUCCESS: The ADD successfully wrote the data to the device • CPP_INVALID_AGENT_HANDLE: No agents correspond to the given handle • CPP_INVALID_PARAMETER: One of the parameters was unrecognized, or would potentially cause overflow or underflow during the data transfer

Table 4: Data Transfer Calls (continued)

Function Call	Item	Description
CppDirectReadMem	Purpose	• ADDs can read directly from an agent's device memory. For performance reasons, the preferred method of data transfer is to DMA data directly from the board. However, for small data transfers it might be more efficient to directly access board memory rather than paying the overhead of setting up a DMA transaction. • The CppDirectReadMem() call reads length bytes from the specified agent's memory buffer starting at an offset of agtMemOffset bytes from the agent's first memory location. The destination of the data is destBuffer. • The addrType parameter indicates what type of address destBuffer is. It can take any of the CppAddrType_t values which cover the possible virtual and physical memory segments.
	Syntax	CppReturnCode_t **CppDirectReadMem** (CppAgtHandle_t agtHandle, CppAddrType_t addrType, uint8_t* destBuffer, uint32_t agtMemOffset, uint32_t length)
	Inputs	• agtHandle: A valid handle to an agent owned by the calling ADD • addrType: Type of address found in destBuffer. The destination can be located in a user-space buffer or a kernel buffer. • destBuffer: The call transfers data from the device to the buffer referenced by destBuffer. • agtMemOffset: Source location of the data to be transferred. The location is given as a byte offset into an agent's shared memory segment. • length: Number of bytes to transfer from the agent's shared memory segment
	Returns	• CPP_SUCCESS: The ADD successfully read the data from the device • CPP_INVALID_AGENT_HANDLE: No agents correspond to the handle given • CPP_INVALID_PARAMETER: One of the parameters was unrecognized or would potentially cause overflow or underflow during the data transfer

Table 4: Data Transfer Calls (continued)

Function Call	Item	Description
CppDirectSetMem	Purpose	• ADDs can fill a range of its memory with a known value. This capability is similar to the standard memset() function. • The CppDirectSetMem() call sets length bytes to fill_value, starting at an offset of agtMemOffset in the agent's memory space.
	Syntax	CppReturnCode_t **CppDirectSetMem** (CppAgtHandle_t agtHandle, uint8_t fillValue, uint32_t agtMemOffset, uint32_t length)
	Inputs	• agtHandle: A valid handle to an agent owned by the calling ADD • fillValue: The unsigned character that is sent to the device memory • agtMemOffset: Destination location of the data to be transferred. The location is given as a byte offset into an agent's shared memory segment. • length: Number of bytes of the agent's memory to set to fillValue
	Returns	• CPP_SUCCESS: The ADD successfully set the memory to fillValue • CPP_INVALID_AGENT_HANDLE: No agents correspond to the handle given • CPP_INVALID_PARAMETER: One of the parameters was unrecognized or would potentially cause overflow or underflow during the data transfer

Table 4: Data Transfer Calls (continued)

Function Call	Item	Description
CppDmaWriteMem	Purpose	• The CppDmaWriteMem() call writes length bytes to the specified agent's memory buffer starting at an offset of agtMemOffset bytes from the agent's first memory location. The source of the data is destBuffer and can reside in either a user-space or kernel-space address location. • The addrType parameter indicates what type of address sourceBuffer is. It can take any of the CppAddrType_t values which cover the possible virtual and physical memory segments. • The CppDmaWriteMem()call will return after inserting the necessary DMA requests into the BDD. The DMA request will complete at an unknown time in the future. Upon completion of the DMA transfer, the BDD will call the agtDrvDmaCallback() routine with the callbackData parameter and the result of the DMA transfer in the status parameter.
	Syntax	CppReturnCode_t **CppDmaWriteMem** (CppAgtHandle_t agtHandle, CppAddrType_t addrType, uint8_t* sourceBuffer, uint32_t agtMemOffset, uint32_t length, void* callbackData, void (*agtDrvDmaCallback)(void *callbackData, CppReturnCode_t status))
	Inputs	• agtHandle: A valid handle to an agent owned by the calling ADD • addrType: Type of address found in sourceBuffer. The source data can be located in a user-space buffer or a kernel buffer. • sourceBuffer: The call will transfer data to the device from the buffer referenced by sourceBuffer. • agtMemOffset: Destination location of the data to be transferred. The location is given as a byte offset into an agent's shared memory segment. • length: Number of bytes to transfer to the agent's shared memory segment • callbackData: The agtDrvDmaCallback()will be called with this parameter upon completion of a DMA transfer • agtDrvDmaCallback: This function will be called upon completion of a DMA transfer
	Returns	• CPP_SUCCESS: The ADD successfully sent a DMA request to the BDD. This does not necessarily mean that the DMA completed. The status of the actual transfer is given as part of the callback routine to the ADD. • CPP_INVALID_AGENT_HANDLE: No agents correspond to the given handle • CPP_INVALID_PARAMETER: One of the parameters was unrecognized, or would potentially cause overflow or underflow during the data transfer

Table 4: Data Transfer Calls (continued)

Function Call	Item	Description
CppDmaReadMem	Purpose	• The CppDmaReadMem() call reads length bytes from the specified agent's memory buffer starting at an offset of agtMemOffset bytes from the agent's first memory location. The destination of the data is destBuffer and can reside in either a user-space or kernel-space address location. • The addrType parameter indicates what type of address destBuffer is. It can take any of the CppAddrType_t values which cover the possible virtual and physical memory segments. • The CppDmaReadMem() call returns after inserting the necessary DMA requests into the BDD. The DMA request will complete at an unknown time in the future. Upon completion of the DMA transfer, the BDD will call the agtDrvDmaCallback() routine with the callbackData parameter and the result of the DMA transfer in the status parameter.
	Syntax	CppReturnCode_t **CppDmaReadMem** (CppAgtHandle_t agtHandle, CppAddrType_t addrType, uint8_t* destBuffer, uint32_t agtMemOffset, uint32_t length, void* callbackData, void (*agtDrvDmaCallback)(void *callbackData, CppReturnCode_t status))
	Inputs	• agtHandle: A valid handle to an agent owned by the calling ADD • addrType: Type of address found in destBuffer. The destination can be located in a user-space buffer or a kernel buffer. • destBuffer: The call transfers data to the device from the buffer referenced by destBuffer. • agtMemOffset: Source location of the data to be transferred. The location is given as a byte offset into an agent's shared memory segment. • length: Number of bytes to transfer from the agent's shared memory segment • callbackData: The agtDrvDmaCallback() is called with this parameter upon completion of a DMA transfer • agtDrvDmaCallback: This function is called upon completion of a DMA transfer
	Returns	• CPP_SUCCESS: The ADD successfully sent a DMA request to the BDD. This does not necessarily mean that the DMA completed. The status of the actual transfer is given as part of the callback routine to the ADD. • CPP_INVALID_AGENT_HANDLE: No agents correspond to the handle given • CPP_INVALID_PARAMETER: One of the parameters was unrecognized or would potentially cause overflow or underflow during the data transfer

Table 4: Data Transfer Calls (continued)

Performance Optimization Calls

These calls are not yet implemented, and are only preliminary.

```
CppReturnCode_t CppMappedDmaWriteMem(
   CppAgtHandle_t AgtHandle,
   CppDmaMapping_t *PageMapping,
   uint32_t PageMapOffset,
   uint32_t AgtMemOffset,
   uint32_t Length,
   void *CallbackData,
   void (*AgtDrvDmaCallback)(void *CallbackData,
     CppReturnCode_t status));

CppReturnCode_t CppMappedDmaReadMem(
   CppAgtHandle_t AgtHandle,
   CppDmaMapping_t *PageMapping,
   uint32_t PageMapOffset,
   uint32_t AgtMemOffset,
   uint32_t Length,
   void *CallbackData,
   void (*AgtDrvDmaCallback)(void *CallbackData,
     CppReturnCode_t status));

CppReturnCode_t CppCreateVirtBufferMap(
   uint8_t *VirtBuffer;
   CppDmaMapping_t *PageMapping);

CppReturnCode_t CppReleaseVirtBufferMap(
   CppDmaMapping_t *PageMapping);
```

Raw Access Calls

These calls are not yet implemented, and are only preliminary.

```
CppReturnCode_t CppGetAgtVirtAddr(
   CppAgthandle_t AgtHandle,
   unsigned long *AgtVirtAddr);
```

Standard Memory Use

Basic Circular Queue Definition

For the purposes of this document, circular queues are contiguous sections of memory in which the top of the Queue (Q_Top) is the lowest memory address, and the bottom of the queue (Q_Bottom) is the highest memory address. Together, head and tail pointers define the free and in-use portion of the queue:

- The **Head** pointer points to the next free memory location.
- The **Tail** pointer points to the first in-use memory location.

  Empty: Q_Tail == Q_Head

  Full: (Q_Head + 1 Mod Queue Size) == Q_Tail

Queues are initialized so that the Q_Head = Q_Tail = Q_Top (an empty condition). All pointers point to qword addresses, as Figure 9 shows.

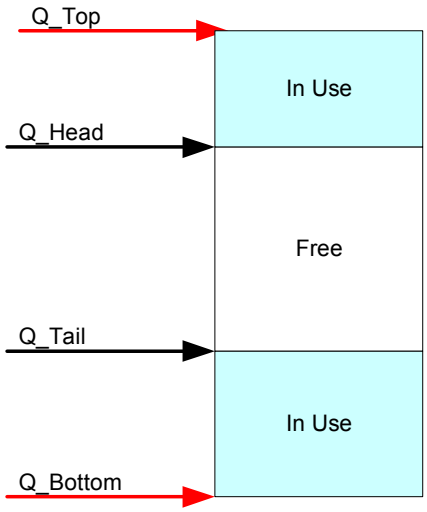

Figure 9: Basic Circular Queue Structure

Queue Types

All hardware and software in-band communications use queues in DDR SDRAM. Three queue types are employed for this purpose:

- **Software** initiates hardware actions using an Input Command Queue.
- **Hardware** supplies completion status using an Output Status Queue.
- **Software/Hardware** data communications use a Data Queue.

While in-band communications occur using these three queue types, queue management takes place using out-of-band communications methods:

- **Software** updates the head pointer of the Input Command Queue and the tail pointer of the Output Status Queue
- **Hardware** updates the tail pointer of the Input Command Queue and the head pointer of the Output Status Queue.
- **Software** controls both the head and tail pointer of the Data Queue.

For more information about out-of-band communications, see “Standard Out-of-Band Communications” on page 33.

Data Queue

Input and output data is stored in the Data Queue. Software manages this queue and dictates input/output data locations to Agents using the aforementioned Input Command Queue. Agents view the Data Queue as a circular queue, bounded by the Top and Bottom pointers that software sets during initialization. If an Agent reaches the bottom of the queue during a read or write operation, it continues the operation at the top of the queue.

Input Header Format

Figure 10 shows the basic format of Input Headers loaded into the Input Command Queue. Input headers have a fixed size of 64 bytes, and only use the first 5 words for software/Agent communications.

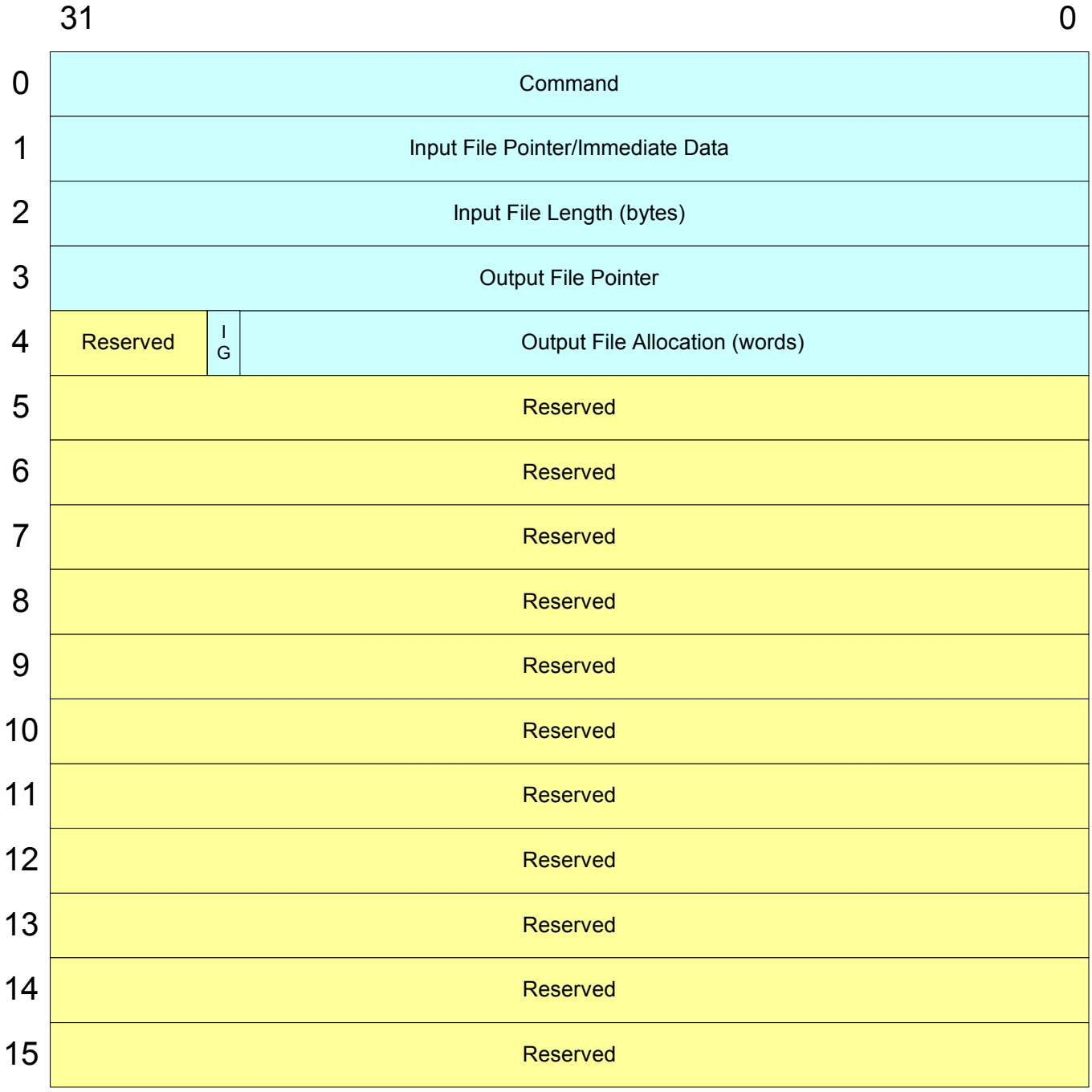

Figure 10: Input Header Basic Format

- The **Command field** has two formats, the Common Command format and the Agent Specific Command format, as described below. Depending on the associated command, the Input File Pointer/Immediate Data field might contain a qword pointer to DDR SDRAM, or 32-bits of data. For commands requiring an Input File Pointer, the Input File Length field contains the length (in bytes) of the associated input file. The Output File Pointer field is a qword pointer to the DDR SDRAM, indicating the 8-byte aligned starting memory location for the Agent's output data. The Output File Allocation Field indicates the maximum amount of memory (in dwords) that the Agent uses for its output data. The Agent must **not** write beyond its allocated memory space.
- The **Interrupt Generate (IG) field** indicates if the Agent is to interrupt software upon the completion of this command.

Common Commands

Common commands are used for all Agents. These commands write to, and read from internal Agent registers, and initialize local Agent memory. All pointers to DDR SDRAM are qword pointers, and all pointers to Agent SRAM are dword pointers. These four commands are currently defined for this Header Format, and the SAI executes them only after all previous jobs complete:

- **Write Register (immediate)**: Copy the contents of the 32-bit Immediate Data field to the register pointed to by the Destination_Address field.
- **Read Multiple**: Copy the contents of the dword aligned memory location pointed to by Source_Start_Address to DDR SDRAM starting at the offset pointed to by the Output File Pointer field; increment the source and destination addresses by 4-bytes; repeat "Output File Allocation" times.
- **Write Multiple**: Copy the contents of the qword aligned DDR SDRAM location starting at the offset pointed to by the Input File Pointer field to the internal memory locations pointed to by the Destination_Start_Address field. Repeat "Input File Length / 4" times.
- **Initialize**: Similar to the Write Multiple Command, except the last 4-bytes of the input file contain a 4-byte checksum. Upon completion of the write portion of the command, the Agent reads back each destination memory location while calculating the 32-bit checksum. The Agent then compares the checksum and reports errors using the Output Header.

Figure 11 through Figure 14 show the specific formats for each of the common commands.

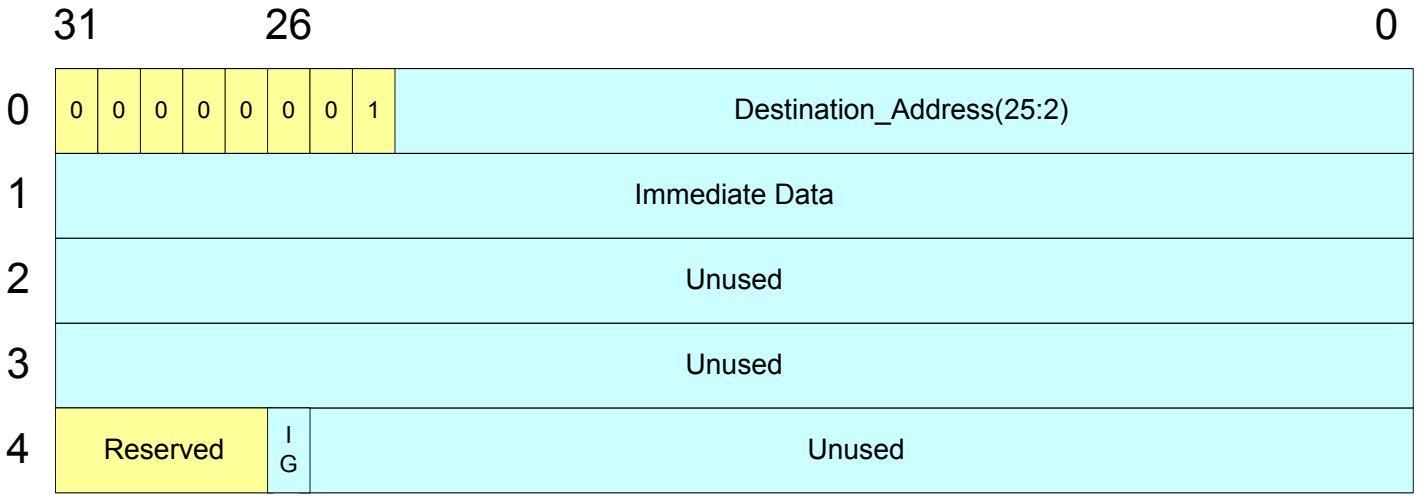

Figure 11: Write Register Command Format

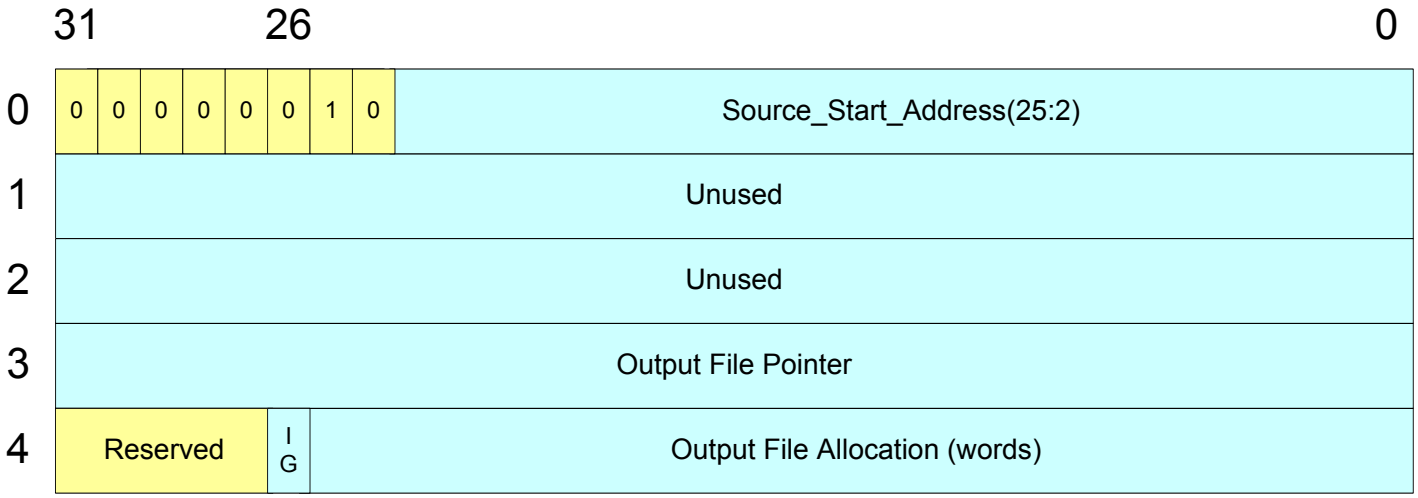

Figure 12: Read Multiple Command Format

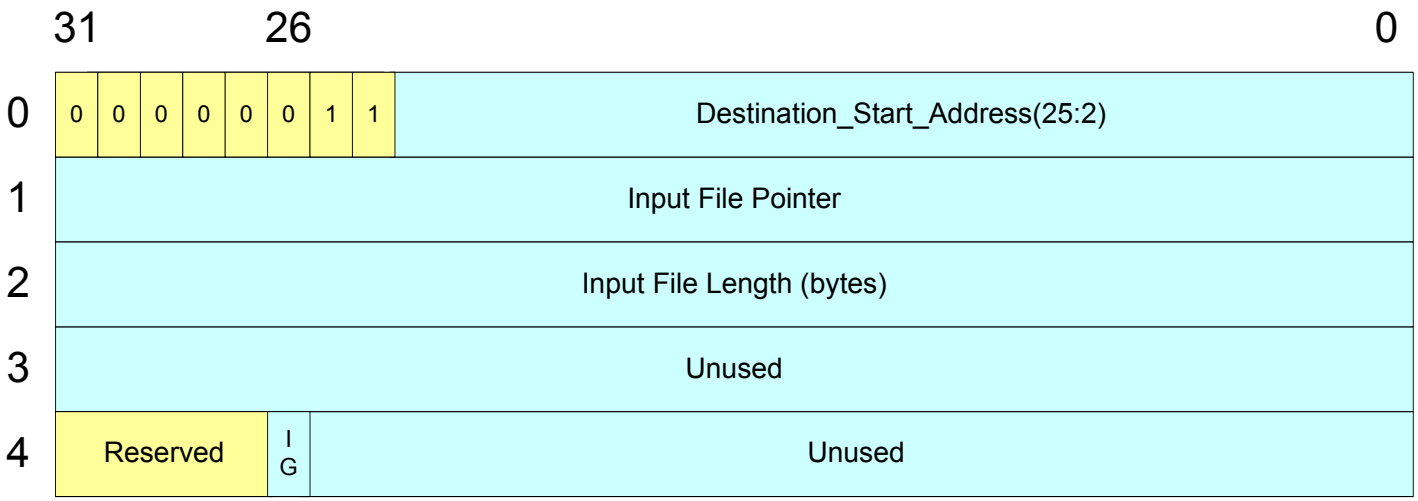

Figure 13: Write Multiple Command Format

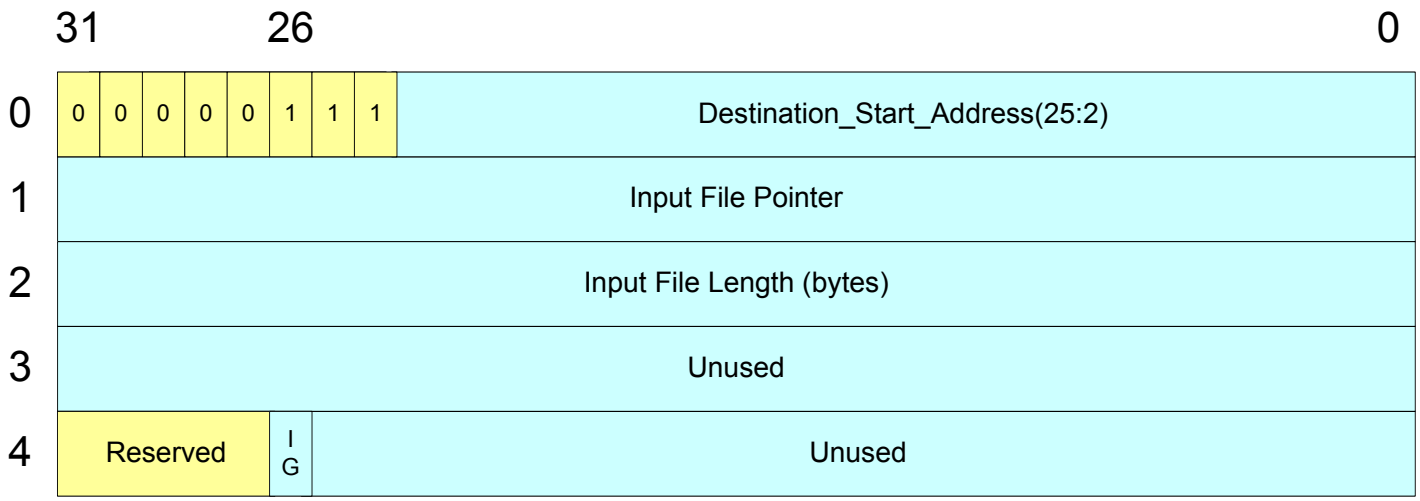

Figure 14: Initialize Command Format

Agent Specific Commands

As Figure 15 shows, the Agent Specific command format is similar to the Common commands. When bit 31 of the first word is set, the Input Header contains an Agent Specific command. Bits 30:16 are unique to the Agent and are defined in the corresponding functional specification. The Job ID field contains an arbitrary 16-bit value unique to the submitted job for the life of that job. Job ID 0x0000 is reserved for Common commands. They have an implied Job ID of 0x0000, and must **not** be used for Agent Specific commands. All pointers are qword pointers.

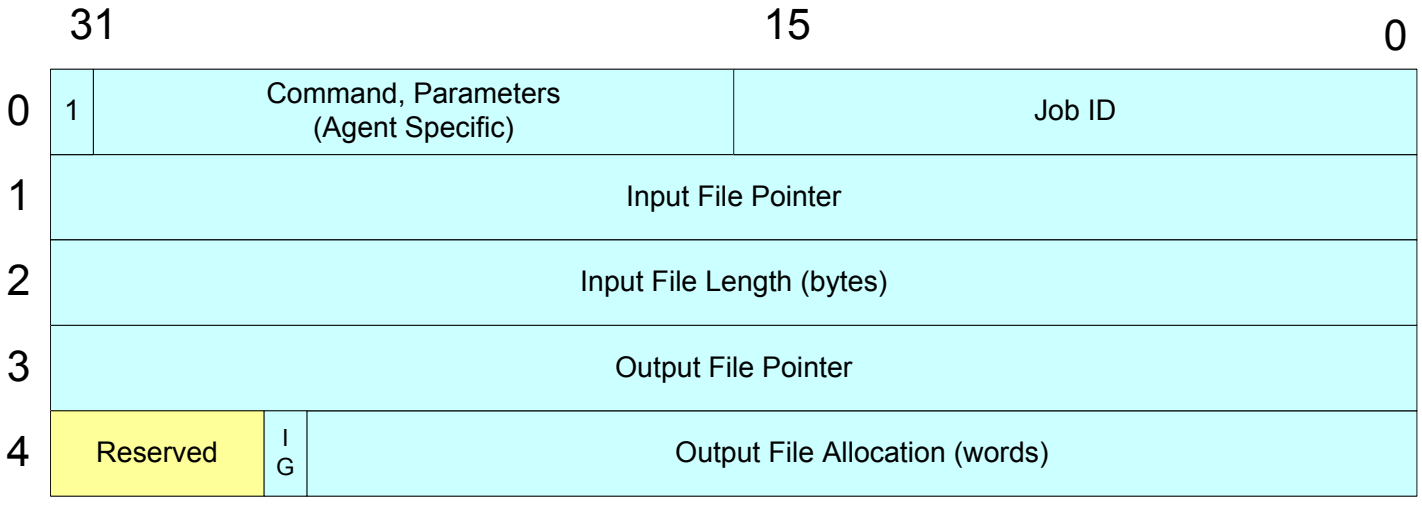

Figure 15: Agent Specific Command Format

Output Header Format

After completing the command retrieved from the Input Command Queue, the Agent writes the resulting status to the Output Status Queue. The output status is contained in 8-byte headers, as Figure 16 shows.

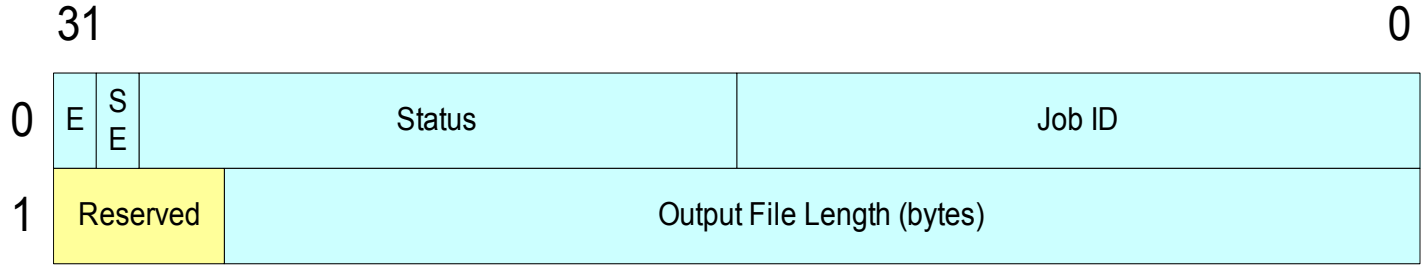

Figure 16: Output Header General Format

Bit 31 of the first word of the Output Header is the Error bit. If the Agent had an abnormal termination of the requested command, it sets bit 31 to 1. If the Agent tries to output more data than there is space allocated by the command descriptor, the SAI sets bits 31:30 to 11. The Status field is 14 bits wide and contains Agent specific status. The lower 16 bits of the first word contain the Job ID, as copied from the Input Header. All Common Commands have an implied Job ID of 0x0000, so all resultant Output Headers contain 0x0000 in the Job ID field. The second dword of the Output Header contains the 28-bit Output File Length, which is the actual number of valid bytes the Agent wrote to DDR SDRAM.

Software uses the Job ID as a reference, to determine the location of the output file in DDR SDRAM. The Output File Length defines the number of bytes to copy to host memory. The SAI guarantees that all Common Commands execute and complete in the order received. Thus, software can determine the location of output files in DDR SDRAM when the Job ID is 0x0000.

Standard Out-of-Band Communications

Out-of-band Communication Channels

Queue management functions, including pointer initialization and pointer updates, occur using out-of-band communications. CPP hardware provides a 32-bit write-only register for software to Agent out-of-band communications and a similar read-only register for Agent to software communications. A second Agent to software register generates a hardware interrupt when updated by the Agent.

The CPP hardware and software Base Driver do not define the format of the two Agent-to-software communication channels. The CPP Base Driver defines a single format for the software-to-Agent channel. As Figure 17 shows, when bit 31 is set, the command is defined as the load DDR SDRAM Base Address. When an Agent receives this command, it sets its internal DDR SDRAM Base Address register to the value contained in bits 30:0. When bit 31=0, the format of the command is user-defined (using the Agent Device Driver). This is a qword address.

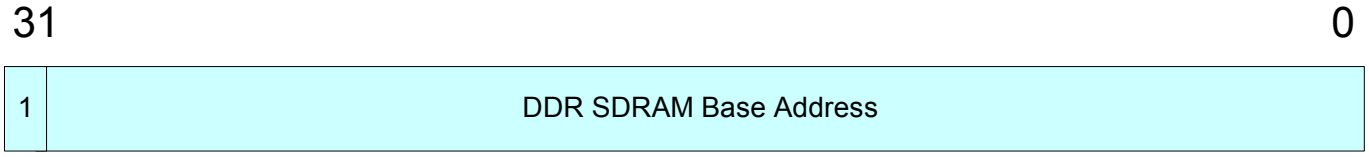

Figure 17: Load DDR SDRAM Base Address Command

Out-of-Band Communication Standard Formats

This section specifies a standard use model for the available out-of-band communications channels, required for the SAI's proper operation.

Basic Format

Figure 18 shows the basic format for Agent/software out-of-band communications. Bit 31 is always 0. Bits 30 through 27 contain the Op Code or command. Except for the Set_DataQ_Top and Set_DataQ_Bottom commands Bits 26 through 24 are reserved. The lower 24 bits contain the pointer or data value associated with the op-code.

All out-of-band communications between software and hardware follow this basic format. Software issues commands to hardware using the software-to-Agent channel. In most cases (exceptions are noted in the next section), hardware echoes software commands through its Agent-to-software channel. Hardware issues pointer updates using the interrupting Agent-to-software channel.

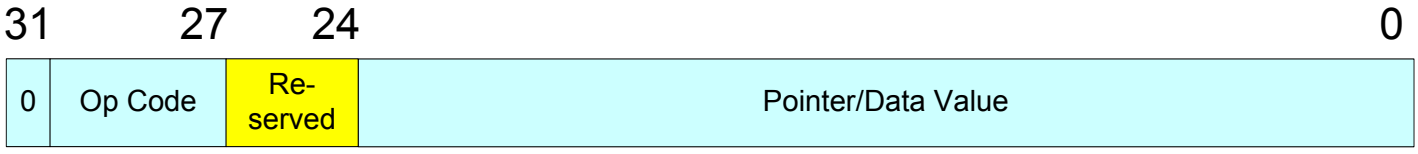

Figure 18: Out-of-Band Communications, Basic Format

Out-of-Band Command Set

Table 5 summarizes the command set. For all commands except Set_DataQ_Top and Set_DataQ_Bottom, pointer sizes are 24 bits. This 8-byte pointer represents 2^{27}, or 128 MB of addressing. This constrains the location of all Input Command Queues and Output Status Queues to the lower 128 MB of allocated space (the physical address equals the Base Address plus the pointer). Because these queues are generally very small compared to the data queue, this constraint has little or no impact upon the design of the Agent Device Driver.

When the Op Code is Set_DataQ_Top or Set_DataQ_Bottom, the reserved field is used for an additional three bits of addressing. Placing the constraint that the Data Queue must start and end on a 64-byte boundary (the lower five bits are always 00000), which allows the Data Queue to be positioned anywhere in the full 64 GB address space.

Op Code	Pointer	Description
0000	Set_Agent_InputQ_Top	Set Agent Input Command Queue Top Pointer
0001	Set_Agent_InputQ_Bottom	Set Agent Input Command Queue Bottom Pointer
0010	Set_Agent_OutputQ_Top	Set Agent Output Status Queue Top Pointer
0011	Set_Agent_OutputQ_Bottom	SetAgent Output Status Queue Bottom Pointer
0100	Update_Agent_InputQ_Head	Update Agent Input Command Queue Head Pointer
0101	Update_Agent_InputQ_Tail	Update Agent Input Command Queue Tail Pointer
0110	Update_Agent_OutputQ_Head	Update Agent Output Status Queue Head Pointer
0111	Update_Agent_OutputQ_Tail	Update Agent Output Status Queue Tail Pointer
1000	Set_DataQ_Top	Set Data Queue Top Pointer
1001	Set_DataQ_Bottom	Set Data Queue Bottom Pointer
1010	TMU_Ping	TMU Ping (TMU Only)
1011	TMU_Reset	TMU Soft Reset (TMU Only)
1100	Agent_Read_Register	Read Internal Memory Mapped Register
1101	SAI_Config	Set SAI Dynamic Configuration
1110	Agent_Config	Set Agent Dynamic Configuration
1111	TMU_Config	Set TMU Dynamic Configuration

Table 5: Out-of-Band Command Summary

Set Input/Output Queue Top/Bottom (Op Codes 0000 – 0001)

Software issues these commands during power initialization, or after a soft reset. A top and bottom pointer pair defines the boundaries of the corresponding circular queue. Although the pointers represent an 8-byte memory address, the Input queue places additional constraints on their alignment. Input queue pointers are 64-byte aligned and the lower 3 bits of the pointer are always 000, as Figure 19 shows. Output queue pointers are 8-byte aligned. For this command, bit 23 is also reserved.

When an Agent receives this command, it echoes it back unchanged through its Agent-to-software channel.

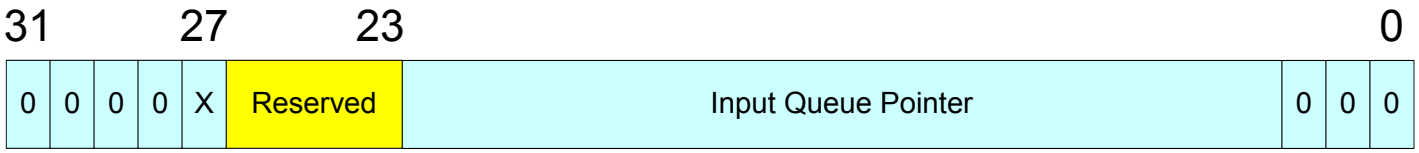

Figure 19: Set_Input Queue Top/Bottom Command Format

Set Output Queue Top/Bottom (Op Codes 0010 – 0011)

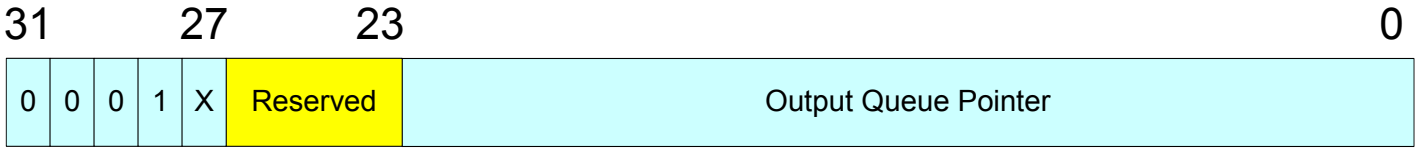

Figure 20: Set_Output Queue Top/Bottom Command Format

Update Input Queue Head (Op Code 0100)

Software issues this command to indicate the presence of additional headers in the Input Command Queue. If the Agent is idle, this command causes it to begin processing the job or jobs that software has placed in the Input Command Queue. If the Agent is currently active, it continues processing until the Input Command Queue is empty, or if it terminates abnormally.

Update Input Queue Tail (Op Code 0101)

Normally, software and hardware do not issue this command. After completing a job, the Agent issues an interrupt with the Update_OutputQ_Head command (described below). This also implies an Update_InputQ_Tail command, because every input header has a corresponding output header. If software issues this command, the SAI updates its internal pointer accordingly, but with undefined results.

Update Output Queue Head (Op Code 0110)

When issued by hardware, this command indicates the presence of additional header(s) in the Output Status Queue. The Agent uses this command to interrupt software upon completion of a job when all interrupt conditions have been met. For more information about interrupt conditions, see “Standard Agent Interface” on page 38.

As each input header has a corresponding output header, this command serves as an implicit Update_InputQ_Tail command. When the Agent updates its internal Output Queue Head pointer, it also updates the corresponding Input Queue Tail Pointer. If the jobs complete out of order, the implied Update_InputQ_Tail command stalls, pending completion of all previously submitted jobs. Software must account for this when moving its copy of the Input Queue Tail Pointer.

If software issues this command, the SAI updates its internal pointer accordingly, but with undefined results.

Update Output Queue Tail (Op Code 0111)

Software issues this command after it has read the contents of the Output Status Queue, usually following an interrupt. This frees the queue for additional output headers.

Set Data Queue Top/Bottom (Op Codes 1000, 1001)

Software issues these commands during initialization or after a soft reset. The top and bottom pair establishes the boundaries of the Data Queue. Figure 21 shows the unique format for these commands.

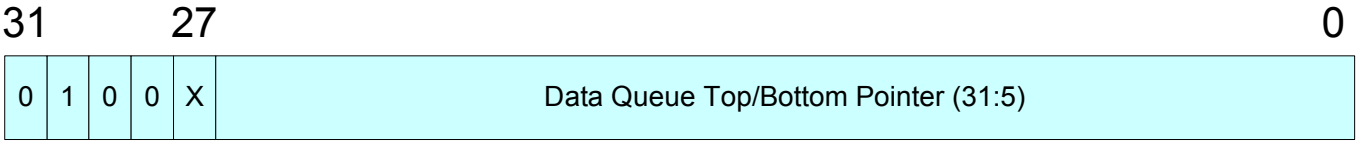

Figure 21: Set_DataQ_Top/Bottom Command Format

Agent Read Register (Op Code 1100)

Software issues this command to discover an Agent's Unique ID, or to read any of its supported internal registers. Rather than echo this command, the Agent responds with the contents of the register pointed to by the Internal Register Address Field, as Figure 22 shows. Bit 23 of the address is reserved for SAI registers. With bit 23 set, the Agent never sees the read register command. The memory map of an Agent is defined in the Agent's functional specification. For consistency, this specification defines the format of the first three memory locations in Table 6. All Agents must follow this format for TMU compatibility.

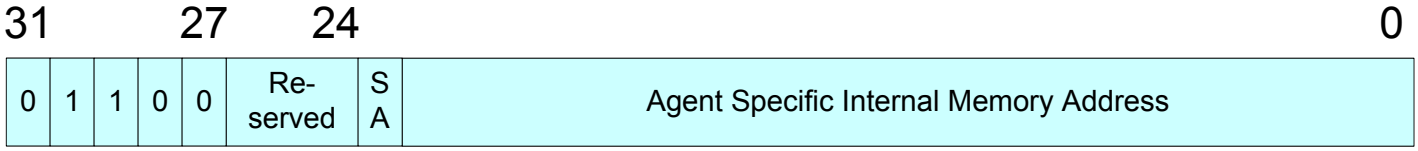

Figure 22: Agent Read Register Command Format

Address	Contents
0x000000	Agent ID: The same ID as placed on the Bit Stream Header. Format is TBD.
0x000001	Major.Minor Revision in BCD format: MMMMmmmm. The TMU defines bits 31:28.
0x000002	Revision date in BCD format: DDMMYYYY

Table 6: Agent Identification Registers, Standard Format

SAI Configuration (Op code 1101)

Software issues this command to setup the SAI's timeout and interrupt threshold counters, and enable the SAI. When the Enable bit is 0, reading or writing to DDR SDRAM by the Agent is disabled. With the Enable bit set to 1, the Agent retrieves and stores headers and data. The Agent Enable has no direct effect on the Agent proper, it simply prevents the retrieval and storage of data from and to DDR SDRAM. SAI Configuration bit 22 is the TMU present bit. When set, the corresponding Agent must use its TMU port for issuing out-of-band I/O writes and interrupts.

The InputQ_Empty timeout field defines the amount of time, after all jobs have completed and the input queue is empty, that the SAI generates an interrupt. This timer decrements at 0.5 MHz intervals.

The interrupt threshold field defines the number of output headers written to the Output Status queue before the Agent issues an interrupting Update_OutputQ_Head command. This provides from 1 to 32 job completions per interrupt. Regardless of the field's contents, the Agent always interrupts if the Input Command queue is empty, or a job abnormally terminates.

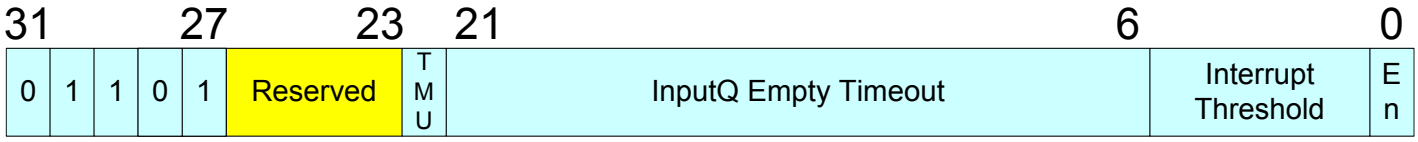

Figure 23: SAI Configuration Format

Acceleration Agent Configuration (Op code 1110)

Each Agent defines the Agent specific mode/configuration field, as Figure 24 shows, in its corresponding functional specification.

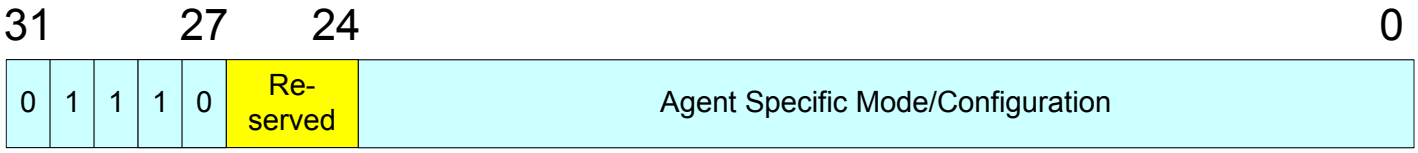

Figure 24: Agent Configuration Command Format

Standard Agent Interface

This section describes the function and interfaces of the SAI. The SAI is a VHDL module that interfaces the low-level Agent Bus Interface and the Agent proper. It provides a simple FIFO interface to the Agent, allowing Agent developers to focus on value added function rather than memory management. The abstraction layer it provides also allows for future enhancements to the memory management scheme without any changes or side effects on the Agent. The SAI follows the memory management and out of band communications protocols established earlier in this document.

General Description

The SAI performs the queue management and software interface functions necessary to move data between DDR SDRAM and the Agent's input and output FIFO. It maintains input and output header queues, located in DDR SDRAM, with a corresponding set of Head and Tail pointers, as described in the CPDK Overview (see "CPDK Overview" on page 3).

When new input headers become available, the SAI reads and interprets the next header. The input header contains a pointer to and the file size of the input file. The SAI then copies the input file from DDR SDRAM into the input FIFO of the Agent. As the Agent performs its assigned operation on the input file, it begins to generate data stored in its output FIFO. The input header also contains a pointer to the corresponding output file, which the SAI uses to copy data from the Agent's output FIFO to DDR SDRAM. After generating and writing the output header to the Output Header Queue, the SAI might interrupt software to indicate the completion of the operation.

This is a step-by-step sequence of events in a typical CPDK job, from a hardware perspective, when using the SAI. The SAI performs steps 6 through 10.

1. Software builds a DMAC descriptor list containing the input data file and the input header.
2. Software initiates a DMAC operation.
3. The DMAC copies the input file from Host Memory to DDR SDRAM.
4. The DMAC copies the input header from Host Memory to DDR SDRAM.
5. The DMAC generates an I/O write to the SAI with an Update_InputQ_Head command.
6. The SAI retrieves the Input Header from DDR SDRAM.
7. The SAI retrieves file data from DDR SDRAM in Agent specified block sizes and copies it to the Agent's input FIFO. When the last word of data is written, as defined by the LOF field of the Input Header, the SAI asserts EOF.
8. If data is available in the Agent's Output FIFO, the SAI copies it to DDR SDRAM in blocks equal to the number of words in the output FIFO, but not to exceed the Agent specified BS_MAX_WR_SZ. The SAI uses the Output File Pointer contained in the Input Header to direct the writes to DDR SDRAM.
9. The SAI continues to write to DDR SDRAM as data becomes available until the Agent asserts EOF, or until the maximum file size, as specified in the Input Header, is reached.

10. If enabled and the Interrupt Threshold is reached, the SAI generates an interrupt to the host using the Update_OutputQ_Head command.
11. Software reads the Output Header from DDR SDRAM (using direct I/O or the DMAC).
12. Software builds a DMAC descriptor list to copy the output file to Host Memory.
13. Software initiates a DMAC operation.
14. The DMAC copies the output file from DDR SDRAM to Host Memory.

Once the system is fully operational, the above sequences overlap to produce a high-throughput pipelined operation. Software does not wait for one operation to complete before submitting another. Similarly, the SAI interleaves DDR SDRAM reads and writes, and retrieves new jobs before existing ones expire.

Interrupts

The SAI asserts an interrupt to notify the host that there are output headers for the host to process, or if the SAI has processed all available commands. These conditions can cause the SAI to interrupt the host, and each interrupt returns the current output queue head pointer, as Figure 25 shows:

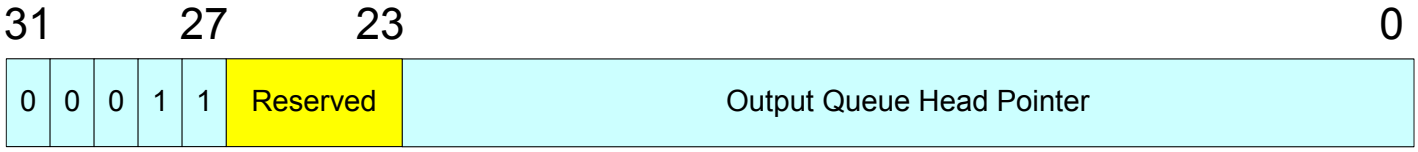

Figure 25: SAI Interrupt

1. Interrupt count equals interrupt threshold
2. Bit 31 set in status returned by the Agent
3. Output file allocation reached
4. Input queue tail == input queue head, command queue empty, input queue empty timer has expired and the interrupt threshold is not hot
5. Output queue head + 1 mod queue size == output queue tail

Linux CPP Architecture

Data Structures

Table 7 lists the Data Structures.

Structure	Item	Description
CppAgtHandle_t	Purpose	• An ADD might request access to an agent configured on a CPP. The CppAgtHandle_t structure is given to an ADD to reference a particular agent. The ADD uses that handle on all future accesses to notify the BDD of which agent it needs to access. • Under Linux, the handle is defined as an integer. The BDD assigns the handles and guarantees that concurrent handles are unique integers.
	Syntax	typedef CppAgtHandle_t int;
CppDmaMapping_t	Purpose	• This structure is not yet implemented, and is only preliminary. • Mapping user-space buffers into an OS kernel for DMA transfers is expensive. As a result, the CPP supports caching buffer mappings to reduce the number of times that mappings must occur. The CppDmaMapping_t structure holds the information required by the BDD to perform a DMA transaction on a user-space buffer.
	Syntax	typedef struct CppDmaMapping { struct kiobuf *iobuf; uint32_t mapping_length; } CppDmaMapping_t;

Table 7: Linux CPP Data Structures

API Calls

The Linux architecture conforms to the BDD to ADD API (see "BDD to ADD Interface" on page 15), with these expectations:

- The CppRegisterAgent() routine currently does not verify that the calling driver has access to the given Agent Identifier before granting access to the agent. As a result, it is possible for multiple ADDs to each own instances of agents with the same agent identifier (although only one driver will have access to a particular agent at a time). This behavior will most likely be changed at some point in the future. As a result, developers must not rely on this behavior.
- The timeout for REMOVE_AGENT is not yet implemented.

Windows* CPP Architecture

This functionality is currently under development, and will be documented here once it becomes available.

Index

D

E

F

G

H

I

K

L

O

P

Q

R

S

T

U

W

Notes